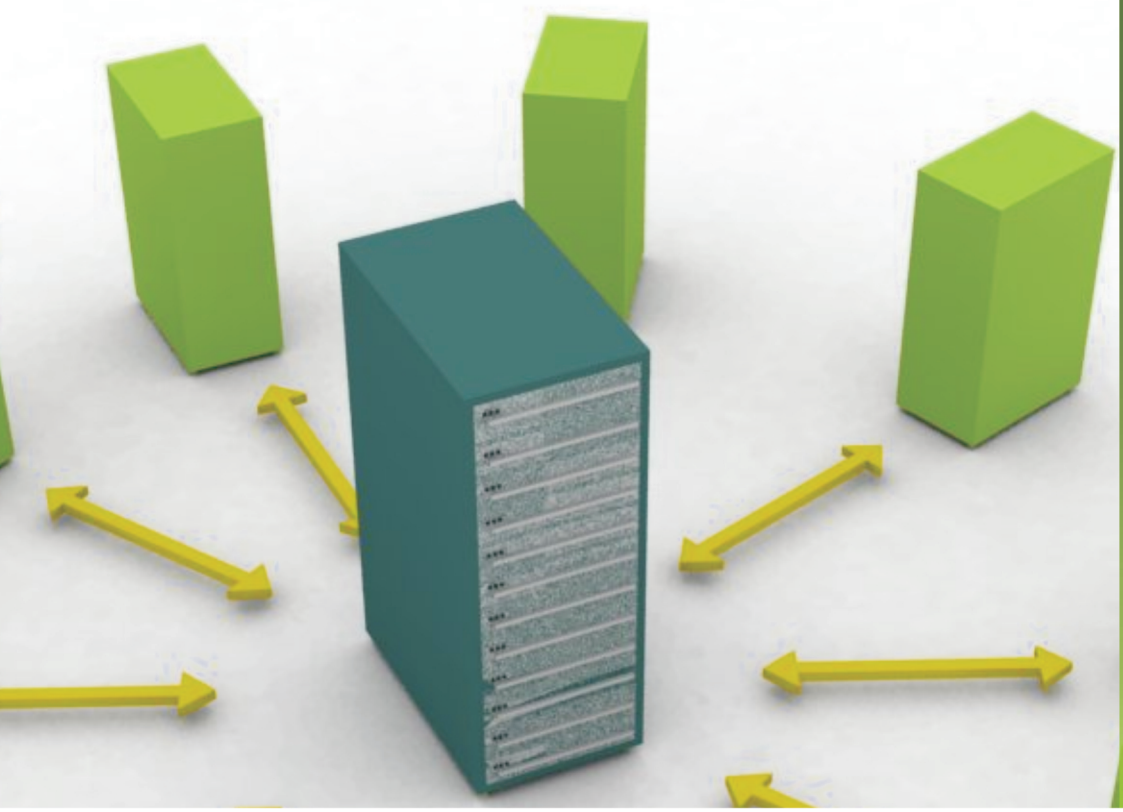
Clube Delphi + PHP

Ano 9 · Edição 106 · R$11,90

DataBase Gear

Rapid SQL: *Quer controle total de seus bancos de dados? Escolha a ferramenta e relaxe.*



Ask the Expert
Acrescentando uma barra de progresso ao copiar um arquivo

QuickUpdate
A evolução da interface de usuário

Rapid SQL
Conheça o Rapid SQL e veja como gerenciar bancos de dados com ele

Boa Idéia
Faça um controle de cheques usando orientação a objetos – Parte II

EasyDelphi
Entenda de forma simples e prática como trabalhar com métodos em Delphi

Expert
Trabalhando com transações avançadas em DBExpress e DataSnap. Aprenda a reaproveitar as transações já abertas pelo DataSnap

Delphi Core
Criando sistemas multicamadas utilizando Borland Socket Server - Parte II

PHP
Construa WebServices em PHP

ISSN 1517990-7

Na Web

Confira no portal ClubeDelphi PLUS 3 vídeo-aulas:

- Análise de pontos de função
- Delphi 2009 – Novidades da VCL e RTL
- Delphi 2009 – Novidades da linguagem

Engenharia de Software
CONFERENCE

22 e 23 de maio – São Paulo

Raras são as oportunidades de ter acesso a informações
que podem transformar sua carreira. Essa é uma delas.

Reserve sua agenda. Inscrições limitadas.
www.devmedia.com.br/es_conference

Maiores informações através do e-mail evento@devmedia.com.br
ou pelo telefone (21) 3382-5025

DevMedia group

Existem momentos onde a estrutura faz toda a diferença.
Se o tempo é ruim, você encontra na Alog um porto seguro. Isto porque sua empresa reduz custos e melhora a governança em TI. Entre em contato e descubra porque a Alog é líder em hosting gerenciado.
REDUÇÃO DOS CUSTOS
AUMENTO DA GOVERNANÇA EM TI
0800 282 3330 | alog.com.br
ALOG
DATA CENTERS DO BRASIL
HOSTING GERENCIADO | COLOCATION | CONTINGÊNCIA | E-MAIL CORPORATIVO | CONECTIVIDADE | SERVIÇOS PROFISSIONAIS

VOCÊ levou anos para desenvolver, testar e colocar seu produto no mercado e **ELE** apenas alguns minutos para **PIRATEÁ-LO.**

SafeNet tem à sua disposição a proteção de software mais avançada e segura. As Chaves de Hardware Sentinel oferecem tecnologia que incorpora criptografia de chave pública, criptografia AES e autenticação interna para garantir o mais alto nível de proteção anti-pirataria.

Peça seu Kit de Desenvolvimento hoje mesmo.

Mais informações www.safenet-inc.com/antipirataria/01
e receba o Estudo de Caso.

SafeNet®
The Foundation of Information Security

Sumário

Olá, eu sou o **DevMan**! Desta página em diante, eu estarei lhe ajudando a compreender com ainda mais facilidade o conteúdo desta edição. Será um prazer contar com sua companhia! Confira abaixo o que teremos nesta revista:

Capa

12 – DataBase Gear e Rapid SQL
Gerencie suas bases de dados em um único ambiente
[Adriano Santos]

Delphi

Boas Práticas, Delphi Core

20 – Borland Socket Server
Criando sistemas multicamadas utilizando Borland Socket Server - Parte II
[*Daniel Sobrinho*]

Boa Idéia

32 – tiOPF na Prática
Faça um controle de cheques usando orientação a objetos - Parte II
[*Paulo Quicoli*]

Banco de Dados

40 – Conhecendo o Rapid SQL
Gerencie de forma simples e organizada suas bases de dados
[*Adriano Santos*]

Easy Delphi

Easy Delphi

46 – Métodos da teoria à prática
Veja na prática como criar métodos em sua aplicação
[*Daniel Sobrinho*]

PHP

PHP

52 – Construa WebServices em PHP
Crie WebServices em PHP
[*Adriano Santos*]

NOVIDADES

EXPERT

56 – Transações avançadas em DBExpress
Aprenda a reaproveitar as transações já abertas pelo DataSnap
[*Daniel Sobrinho*]

Você percebeu os ícones ao lado de cada matéria? Eles indicam o que você vai encontrar no artigo – dessa forma, você também pode ter uma idéia geral do que vai encontrar nesta edição como um todo! Os editores trabalham sempre no sentido de fechar a revista seguindo esta definição, para oferecer a você o melhor conteúdo didático!

Confira abaixo a lista com a definição dos tipos de artigo encontrados nesta edição:

[Boas Práticas] Um dos objetivos da revista é levar para o leitor não somente as melhores técnicas mas também as melhores práticas. Esse tipo de artigo foca em técnicas que poderão aumentar a qualidade do desenvolvimento de software.

[Boa Idéia] A esse tipo de artigo é quase um artigo "Mão na Massa" - a diferença é que o exemplo do artigo é tão criativo que é considerado uma "Boa Idéia", dentro do contexto do desenvolvimento de software.

[Expert] Artigo com foco no leitor avançado.

[Novidades] Artigos sobre tecnologias de ponta, que ainda não fazem parte o dia-a-dia do desenvolvedor mas que prometem ser a próxima febre.

[Easy Delphi] Com foco no desenvolvedor iniciante.

[PHP] Artigo com foco no PHP puro.

Clube Delphi +PHP

Ano 9 - 106ª Edição 2009 - ISSN 1517990-7 Impresso no Brasil

Corpo Editorial

Editor Geral
Adriano Santos
adriano.santos@devmedia.com.br

Editor Técnico
Paulo Quicoli
pauloquicoli@devmedia.com.br

Comissão Editorial
Fabrício Desbessel, Paulo Quicoli, Adriano Santos e Guinther Pauli.

Diagramação
Romulo Araujo
romulo@devmedia.com.br

Revisão e Supervisão
Thiago Vincenzo - thiago.v.ciancio@devmedia.com.br

Distribuição
Fernando Chinaglia Dist. S/A
Rua Teodoro da Silva, 907
Grajaú - RJ - 206563-900

Atendimento ao Leitor

A DevMedia conta com um departamento exclusivo para o atendimento ao leitor. Se você tiver algum problema no recebimento do seu exemplar ou precisar de algum esclarecimento sobre assinaturas, exemplares anteriores, endereço de bancas de jornal, entre outros, entre em contato com:

Carmelita Mulin – Atendimento ao Leitor
www.devmedia.com.br/central/default.asp
(21) 3382-5025

Kaline Dolabella – Gerente de Marketing e Atendimento
kalined@terra.com.br
(21) 3382-5025

Publicidade

Para informações sobre veiculação de anúncio na revista ou no site e para fechar parcerias ou ações específicas de marketing com a DevMedia, entre em contato com:

Kaline Dolabella
publicidade@devmedia.com.br

Fale com o Editor!

É muito importante para a equipe saber o que você está achando da revista: que tipo de artigo você gostaria de ler, que artigo você mais gostou e qual artigo você menos gostou. Fique a vontade para entrar em contato com os editores e dar a sua sugestão!
Se você estiver interessado em publicar um artigo na revista ou no site ClubeDelphi, entre em contato com os editores, informando o título e mini-resumo do tema que você gostaria de publicar:

Adriano Santos - Editor da Revista
adriano.santos@devmedia.com.br

EDITORIAL

Como você cria, monitora, presta manutenção e faz alterações em banco de dados de clientes? Aliás, a pergunta certa é: como você lida com banco de dados no dia-a-dia? Conheço muitos desenvolvedores que mantém versões de seus bancos atualizadas em seus servidores, então inserem, alteram e excluem objetos no banco de dados e em seguida guardam os *scripts* para mais tarde efetuar as devidas alterações nos clientes. Outros desenvolvem funções e *units* próprias para isso na aplicação, assim o sistema se atualiza automaticamente. E outros nem tem tanta intimidade assim como *BD's*. Para vencer esses obstáculos a *Embarcadero Technologies* desenvolveu vários programas especialmente para trabalhar com banco de dados. Alguns desses programas são específicos para *DBA's* e outros para o programador. Eu vi de perto todos eles e resolvi mostrá-los através de um artigo. Confesso que são programas bem interessantes e vale a pena dar uma conferida de perto. Ainda mostro detalhadamente como funciona o aplicativo *Rapid SQL* também da *Embarcadero*.

Nessa edição veremos a finalização da aplicação para controles de cheques do Paulo Quicoli, dessa vez veremos como pesquisar os cheques na base de dados.

O Daniel Sobrinho também finaliza seu mini-curso de *BSS (Borland Socket Server)* demonstrando como criar a aplicação cliente e fazendo-a acessar o servidor de aplicações. Em seu segundo artigo, veremos boas práticas na criação de métodos em nossa aplicação. Por fim, ele explica de forma avançada alguns macetes para criação de transações no *dbExpress*.

Em *PHP* confiram meu artigo que mostra como trabalhar com *WebServices* nessa linguagem. Criação do servidor e da aplicação cliente. Veremos duas maneiras distintas de se trabalhar com essa tecnologia. Certamente, aprender a trabalhar com *WebServices* hoje em dia é estar em sintonia com a tecnologia.

Espero que gostem da edição. Boa leitura e até a próxima.

Adriano Santos

adriano.santos@devmedia.com.br

Editor Chefe da revista Clube Delphi
Conheça nosso BLOG em *clubedelphiphp.blogspot.com*

ClubeDelphi PLUS

DevMedia group

A revista ClubeDelphi é parte integrante da assinatura ClubeDelphi PLUS.
Para mais informações sobre o pacote PLUS, acesse:
http://www.devmedia.com.br/clubedelphi/portal.asp

SISFÁCIL 2.0

FICOU AINDA,

MAIS FÁCIL

ADMINISTRAR SEU SITE

Enquete, Fotos, Notícias, Blogs e muito mais.

DISPONÍVEL PARA CLIENTES RJHOST

Seu site **pronto** e **completo**

www.sisfacil.com.br

RJHOST

ClubeDelphi PLUS

Informativo ClubeDelphi

Portal ClubeDelphi

www.clubedelphi.net/portal

+ 600 vídeo aulas e 7 cursos online

Brinde na web desta edição - 3 vídeos! Confira:

3 Vídeos

• **Análise de pontos de função:** Veja com o Marco Antônio P. Araújo como dimensionar, estimar, os requisitos que serão implementados ou mantidos por seu sistema aplicando essa técnica de métrica de software.
http://www.devmedia.com.br/articles/viewcomp.asp?comp=11292&hl=

• **Delphi 2009 – Novidades da VCL e RTL:** Descubra junto com Rodrigo Carreiro Mourão os novos controles que foram adicionados no Delphi 2009 e quais classes base que receberam novos métodos.
http://www.devmedia.com.br/articles/viewcomp.asp?comp=12064&hl=

• **Delphi 2009 – Novidades da linguagem:** Nesta vídeo aula Rodrigo Carreiro Mourão mostra as novidades da linguagem Delphi, como generics e métodos anônimos.
http://www.devmedia.com.br/articles/viewcomp.asp?comp=12063&hl=

Para acessar os vídeos utilize os seguintes dados

Login: **DVM.CD** *Senha:* **dXk42v**

Gostou das vídeo aulas? O portal *www.devmedia.com.br* possui mais de **2 mil vídeo aulas** e **dezenas de cursos online** sobre desenvolvimento de software! Agora você pode comprar as vídeo aulas que preferir e fazer sua própria combinação de vídeos! Saiba mais em *www.devmedia.com.br/creditos*

Edições Anteriores da ClubeDelphi

Você pode comprar todas as edições anteriores da ClubeDelphi através do site da DevMedia!

Para isso basta acessar *https://seguro.devmedia.com.br/edicoes_anteriores.asp*

Dê seu feedback sobre esta edição!

A ClubeDelphi tem que ser feita ao seu gosto. Para isso, precisamos saber o que você, leitor, acha da revista!

Dê seu voto sobre esta edição, artigo por artigo, através do link:

www.devmedia.com.br/clubedelphi/feedback

Para votar, você vai precisar do código e senha de banca desta edição, que é:
Login: **DVM.CD** Senha: **dXk42v**

Hospede seus sites de forma simples, rápida e ilimitada.

Plano **Básico**

- 1 domínio principal
- 50 domínios hospedados
- Transferência **ilimitada**
- Espaço **ilimitado**
- 10 contas de e-mail
- PHP, ASP e ASP.NET
- Painel de controle Plesk
- Suporte 24hs

Tudo isso por apenas

R$ **15,90** por mês
pagamento trimestral

Assine agora mesmo e descubra por que Hospedix é a hospedagem de sites preferida entre os profissionais.

- Conheça também nossos planos profissionais com registro de domínio grátis e isenção de taxas anuais enquanto for cliente.

HOSPEDIX
HOSPEDAGEM PROFISSIONAL

www.hospedix.com.br

30 DIAS GRÁTIS

Ao assinar, digite o código de desconto exclusivo para leitores desta revista e ganhe 30 dias inteiramente grátis.

RVSTMEDIA

Perguntas e Respostas

Acrescentando uma barra de progresso ao copiar um arquivo

Perguntas respondidas por Paulo Quicoli
pauloquicoli@devmedia.com.br

Nosso leitor Eduardo Gomes precisa exibir de forma gráfica a cópia de arquivos, utilizando uma barra de progresso para exibir o quanto do arquivo que já foi copiado. Nós aqui da ClubeDelphi vamos te ajudar Eduardo.

Como é sabido, a *API* do *Windows* nos disponibiliza a função *CopyFile* que permite a cópia de um arquivo para um determinado lugar. Seu uso em *Delphi* seria como a seguir:

```
CopyFile('C:\ArquivoACopiar.txt','D:\CopiaDoArquivoACopiar.
txt',false);
```

O primeiro parâmetro é o arquivo a ser copiado e o segundo é o seu destino, incluindo nome do arquivo. O último parâmetro indica se, caso o destino já exista, a cópia seja interrompida. Essa API funciona perfeitamente, porém ela não disponibiliza meios de saber a porcentagem já copiada do arquivo. A solução então é realizar a cópia de outra maneira. Vamos realizar nós mesmos uma cópia manual. Crie uma nova aplicação *Win32* e na seção *private* do formulário crie uma *procedure* chamada *CopiarArquivo(Origem, Destino: string; BarraProgresso: TGauge)*. Implemente essa *procedure* como na **Listagem 1**.

Listagem 1. Copiando um arquivo de forma manual

```
01 procedure TForm1.CopiarArquivo(Origem, Destino: string;
02 BarraProgresso: TGauge);
03 const
04   TamanhoBufferLeitura = 16384;
05 var
06   StreamOrigem,
07   StreamDestino: TFileStream;
08   PonteiroBuffer: Pointer;
09   QuantidadeLida: Integer;
10   TotalLido, X, TamanhoArquivoOrigem: LongInt;
11 begin
12   TotalLido := 0;
13   StreamOrigem := TFileStream.
      Create(Origem, fmOpenRead or
14    fmShareDenyWrite);
15
16   TamanhoArquivoOrigem := StreamOrigem.size;
17
18   with BarraProgresso do
19   begin
20     MinValue := 0;
21     MaxValue := 100;
22     Progress := 0;
23   end;
24
25   try
26     StreamDestino := TFileStream.
        Create(Destino, fmCreate or
27      fmShareExclusive);
28     try
29       GetMem(PonteiroBuffer,
          TamanhoBufferLeitura);
30       try
31         QuantidadeLida := StreamOrigem.
            Read(PonteiroBuffer^,
32          TamanhoBufferLeitura);
33         QuantidadeLida := StreamDestino.
            Write(PonteiroBuffer^,
34         QuantidadeLida);
35
36         TotalLido := TotalLido +
            QuantidadeLida;
37         while (QuantidadeLida > 0) do
38         begin
39           Application.ProcessMessages;
40           QuantidadeLida := StreamOrigem.
              Read(PonteiroBuffer^,
41            TamanhoBufferLeitura);
42           QuantidadeLida := StreamDestino.
              Write(PonteiroBuffer^,
43            QuantidadeLida);
44           TotalLido := TotalLido +
              QuantidadeLida;
45           BarraProgresso.Progress :=
              Trunc((TotalLido /
46             TamanhoArquivoOrigem)*100);
47           BarraProgresso.Update;
48         end;
49       finally
50         FreeMem(PonteiroBuffer,
            TamanhoBufferLeitura);
51       end;
52     finally
53       StreamDestino.Free;
54     end;
55   finally
56     StreamOrigem.Free;
57     BarraProgresso.Progress := 0;
58   end;
59 end;
```

Basicamente ela transfere o conteúdo de um arquivo para outro, ou seja, ela recebe um arquivo de origem e um caminho para destino. Essa *procedure* cria um arquivo vazio como destino e transfere os bytes do arquivo de origem para esse novo arquivo. Se analisarmos bem, isso não é uma cópia e sim uma *clonagem* de arquivo. Vamos explorar o que essa *procedure* exatamente está fazendo.

Como vamos ler o arquivo de origem de forma manual, é necessário determinar o quanto que vamos ler em cada leitura. Isso é definido na constante *TamanhoBufferLeitura*, que em nosso caso está representando o tamanho de 16Kbytes. Ou seja, vamos ler um arquivo em partes de 16Kb. Posteriormente na linha 13 nós abrimos para leitura o arquivo de origem e, através do parâmetro *fmShareDenyWrite* bloqueamos sua escrita. Isso significa que se o arquivo que vamos copiar for aberto por outro processo, qualquer tentativa de se alterar o arquivo será bloqueada.

Na linha 14 obtemos o tamanho total do arquivo de origem. Esse valor será utilizado para o cálculo do progresso da cópia, exibido em um *TGauge*. Depois criamos o arquivo de destino deixando em modo exclusivo, assim ninguém poderá abri-lo enquanto a cópia é realizada.

Na linha 31 lemos o arquivo de origem, armazenando os 16kb lidos na variável *PonteiroBuffer* e depois passamos esses mesmos 16kb para o arquivo de destino através do método *StreamDestino.Write*. Essa é a primeira leitura do arquivo. Então na variável *TotalLido* vamos acumulando o total de *bytes* já lidos. Esse valor também será utilizado para o cálculo do progresso da cópia. Em seguida, montamos um *loop* que irá ler o arquivo até o fim. Veja que no *while* verificamos se *QuantidadeLida* é maior que zero, isso porque o retorno do método *StreamDestino.Write* indica o quanto foi gravado, se for zero, significa que o arquivo chegou ao seu final. Então mais uma vez acumulamos em *TotalLido* a quantidade de *bytes* que já foram lidos e na linha 45 fazemos um cálculo simples de porcentagem para atualizar o *TGauge* passado como parâmetro. Por fim, liberamos os recursos alocados.

Agora, basta utilizar essa *procedure* passando o arquivo de origem, o arquivo de destino inclusive com seu nome e o *Gauge* que será responsável em exibir o progresso.

Criando paginação em Web com VCL for Web

Nós já vimos paginação em diversos artigos nos últimos tempos, Delphi for .net e PHP, por exemplo. Mas como fazer isso em VCL for Web, antigo IntraWeb? Essa é a pergunta do nosso colega Otávio Menezes.

Bem, a explicação é mais simples do que podemos imaginar. A maneira mais fácil de fazer isso é utilizar o método MoveBy do DataSet. Vamos a um exemplo.

Inicie uma nova aplicação usando o menu File>New>Other>Delphi Projects>VCL for the Web>VCL for the Web Application Wizard e utilize a opção StandAlone Application. Salve o projeto em uma pasta de sua preferência e em seguida crie uma conexão com um banco de dados usando os componentes que acesso que desejar. Em nosso caso usaremos um SQLConnection, SQLDataSet, DataSetProvider e um ClientDataSet. Insira também um DataSource e um IwDbGrid. Não entraremos em detalhes quanto as conexões, apenas tenha ciência de como criar uma aplicação simples que tenha como objetivo listar uma quantidade de registros no IwDbGrid.

Modifique a propriedade RowLimit do IwDbGrid inserindo uma quantidade de registros que desejar que o sistema liste para você, por exemplo: de 100 em 100 registros. A única coisa que precisaremos fazer agora para paginar é incluir dois botões, um para trazer os próximos registros e outro para os registros anteriores.

No código do botão Registros Anteriores digite o código a seguir:

```
IwDbGrid1.DataSource.DataSet.MoveBy(-IwDbGrid1.RowLimit);
```

E no botão Próximos Registros digite o seguinte código:

```
IwDbGrid1.DataSource.DataSet.MoveBy(IwDbGrid1.RowLimit);
```

Perceba que usamos a propriedade RowLimit para indicar como se comportará o método MoveBy. Esse método moverá o cursor para os próximos 100 registros (assim como configuramos na propriedade) ou voltará para os 100 anteriores.

Um detalhe é que o método MoveBy está associado ao DataSet, logo é possível utilizá-lo em DbGrid comum para aplicações Win32/Desktop. ●

Dê seu feedback sobre esta edição!

A Clubedelphi tem que ser feita ao seu gosto. Para isso, precisamos saber o que você, leitor, acha da revista!

Dê seu voto sobre este artigo, através do link:
www.devmedia.com.br/clubedelphi/feedback

Quick Update

A evolução da interface de usuário

Paulo Quicoli

pauloquicoli@devmedia.com.br

Editor técnico da revista ClubeDelphi e .Net Maganize, formado em processamento de dados pela FATEC-TQ. Atua como analista e desenvolvedor da ControlM Informática (www.controlm.com.br). Utiliza Delphi desde sua primeira versão, entusiasta do desenvolvimento orientado a objetos tem publicado vários artigos sobre o assunto.Blog: www.pauloquicoli.spaces.live.com

Em 1996 entrei oficialmente para área de informática e concluí meu curso técnico de processamento de dados. De lá para cá pude ver a evolução das interfaces de usuário e como sua exigência tem aumentado. Recentemente visitando o site do *Mix09* (*www.visitmix.com*), que é um evento da *Microsoft* onde são exibidas as novidades para *web*, encontrei um vídeo (*http://videos.visitmix.com/MIX09/C30M*) muito interessante relacionado a essa evolução citada. Quero expor aqui alguns pontos interessantes sobre o assunto.

Vamos comparar as três *ondas* existentes, posso dizer que passei por duas elas. Observe a **Figura 1**.

Qualquer garoto de 16 ou 18 anos que começa agora na área de desenvolvimento de software para *Windows* consegue imaginar que essa era a interface predominante para operar um computador? Essa foi a primeira *onda*. Podemos dizer que era basicamente baseada em texto e a experiência do usuário era praticamente *desconectada* da interface. Esse tipo é chamado de *Command Line Interface*, ou interface de linha de comando.

Na segunda onda já temos um avanço, a *Graphic User Interface*, ou interface gráfica de usuário, como mostram as **Figuras 2** e **3**.

A *GUI* trouxe uma melhor experiência oferecendo gráficos que possibilitam o *reconhecimento visual* por parte do usuário. É nesta onda que nos encontramos. Procuramos sempre utilizar ícones, imagens, gráficos que favoreçam essa percepção do usuário. Contudo, sua experiência ainda é indireta pois temos um *mouse* e um *teclado*.

A próxima *onda* que vamos enfrentar virá junto com o lançamento do *Windows 7*. É a *Natural User Interface*, ou interface natural do usuário, como pode ser visto na **Figura 4**.

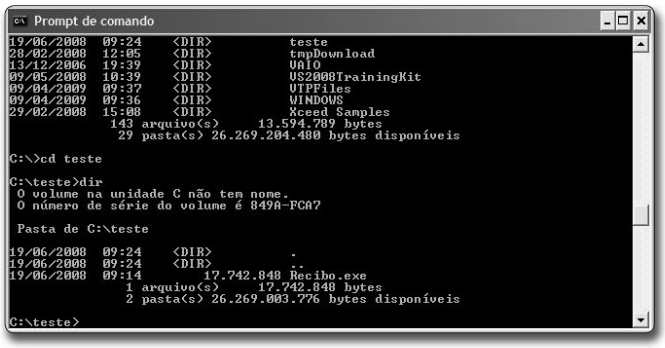


Figura 1. Prompt de comando

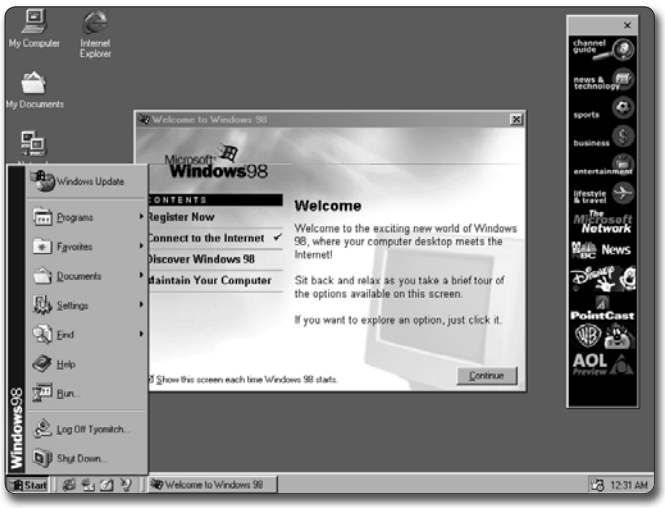


Figura 2. Windows 98

Na verdade já existem vários dispositivos dessa *nova onda*, como alguns celulares sensíveis ao toque e a máquina *Surface* da *Microsoft* (*www.microsoft.com/surface*). Porém a disseminação dessa tecnologia se dará quando o usuário final tiver isso no computador de sua casa ou escritório. Sua vantagem no quesito *experiência de uso* em relação às outras está no fato de que o usuário estará manipulando *objetos* que reagem ao toque, mover. A intuição é aplicada nessa interface, pois existe uma interação direta com o sistema.

Preparando-se para a mudança

Com uma demanda de novas interfaces é preciso que sua empresa se prepare para apresentar, além de um sistema funcional, um sistema bonito. Por isso existem alguns passos que visam sistematizar o desenvolvimento de interfaces, chamado de *ciclo de desenvolvimento de interface:*

• ***Passo 1: Idéia conceitual*** - É onde se deve *rabiscar* as telas, liberando imaginação para sua composição, levando em conta também a interação do usuário. Nessa fase, qualquer integrante da equipe pode participar, incluindo o próprio cliente. Tudo isso para que idéias sejam trocadas para se conseguir no final uma inovação;

• ***Passo 2: Design de interação*** - É a formalização da interação, dos passos, que o usuário deve percorrer para realizar determinada função. Isso é representado por meio de diagramas. Aqui é representado como cada controle visual irá se comportar, reagindo ao que o usuário faz durante a interação;

Figura 3. Windows Vista

Figura 4. Microsoft Surface

• ***Passo 3: Design visual*** - Baseando-se na documentação gerada pelos passos anteriores, é criado então o visual do sistema na forma de protótipos que podem ser revistos com a equipe até mesmo o cliente;

• ***Passo 4: Produção*** - Com o visual aprovado, passa-se a integrar oficialmente os visuais na aplicação;

• ***Passo 5: Acabamento*** - Com uso, vê-se coisas que não foram pensadas, então neste passo refinamos o design.

Independentemente da linguagem de programação é preciso desenvolver uma *tela legal* para nossos sistemas, portanto, fica aqui a dica. ●

Dê seu feedback sobre esta edição!

A Clubedelphi tem que ser feita ao seu gosto. Para isso, precisamos saber o que você, leitor, acha da revista!

Dê seu voto sobre este artigo, através do link:

www.devmedia.com.br/clubedelphi/feedback

Nesta seção você encontra artigos intermediários sobre Delphi Win 32 e Delphi for .NET

DataBase Gear

Conheça as principais ferramentas da Embarcadero para auxiliar o uso de Bancos de Dados

Nesse artigo veremos
- Download e instalação do RapidSQL;
- O que é o RapidSQL;
- Como utilizá-lo na prática.

Qual a finalidade
Apresentar a ferramenta e mostra como utilizá-la no dia-a-dia para manutenção de bancos de dados;

Quais situações utilizam esse recurso?
A todo momento a ferramenta pode ser utilizada. Tanto para a criação quanto manutenção de bancos de dados é recomendado que se use alguma ferramenta front-end

Adriano Santos
falecom@adrianosantos.pro.br
É desenvolvedor Delphi desde 1998, especialista em aplicações Win32, professor e programador PHP, bacharel em Comunicação Social pela Universidade Cruzeiro do Sul, SP. É Editor Chefe da revista ClubeDelphi e Editor Geral da parte .net da WebMobile. Atua como Gerente de TI na SoftPark Informática e membro fundador do DUG-SP (www.dug-sp.com). Mantenedor do blog Delphi to Delphi (www.delphitodelphi.blogspot.com) com dicas, informações e tudo sobre o desenvolvimento Delphi.

Resumo do DevMan

Já sabemos que o desenvolvimento de sistemas não se limita apenas ao uso de IDE's, isso porque grande parte das vezes desenvolvemos aplicações que utilizam-se de bancos de dados. É certo também que uma boa parte dos programadores utiliza-se de algum software para fazer o gerenciamento e a manutenção do banco de dados da aplicação.

Recentemente a Embarcadero Technologies adquiriu a linha de compiladores da Borland que tinha como principais produtos o JBuilder, Delphi, Delphi for PHP, C++ Builder e o Interbase. Para quem não sabe, a Embarcadero possui uma ampla gama de software para gestão de bancos de dados. Nesse artigo, veremos alguns dos principais aplicativos da empresa e como eles funcionam e nos auxiliam no dia-a-dia.

Gerenciar o banco de dados de uma aplicação nem sempre é uma tarefa do desenvolvedor. Mas também nem sempre o desenvolvedor tem a mão um *DBA* para fazer esse trabalho. Além do mais, quase sempre a tarefa de gerar consultas mais rápidas e evitar gargalos na rede acaba gerando conflitos entre ambos profissionais. Entretanto, a fronteira entre o *DBA* e o desenvolvedor pode ser facilmente

rompida utilizando-se de ferramentas para a criação, gerenciamento, manutenção e otimização do banco.

Além do mais, todos nós desenvolvedores sabemos que um bom programador precisa ao menos saber alguns princípios básicos e até intermediários de como gerenciar o *BD*. Pensando nisso a *Embarcadero Technologies*, atual proprietária dos compiladores da *Borland*, desenvolveu uma série de aplicativos com os mais variados recursos para os mais variados tipos de usuários. Com a aquisição feita recentemente, a linha de ferramentas para banco de dados passou a chamar-se *DataBase Gear*. São diversas ferramentas de gestão e manutenção de bancos de dados.

O leque de produtos da empresa inclui ferramentas para manutenção, gerenciamento, monitoramento, monitoramento de mudanças e até ferramentas de modelagem.

Selecionei nesse artigo as principais aplicações e recursos que vi em um treinamento que participei em São Paulo com o pessoal da *Embarcadero*. Ao longo das próximas edições, mostrarei outras ferramentas e conseqüentemente outras funcionalidades.

Apresentando as ferramentas

Com mencionei, são diversas as ferramentas que a empresa possui. Não há como mostrar todas nesse artigo por serem ferramentas complexas e com diversas funcionalidades. Portanto, farei um breve resumo de algumas delas para que possamos entendê-las e conhecê-las. Os aplicativos comentados aqui serão:

• *Rapid SQL:* Essa ferramenta pode ser utilizada pelo desenvolvedor para criar *bancos de dados, tabelas, campos, views* e ainda dar manutenção no *BD*. É possível conectar-se aos mais diversos bancos de dados via conexão nativa e/ou *ODBC Driver*;

• *DB Optimizer:* Essa é com certeza uma das ferramentas mais interessantes. É possível fazer uma otimização do bando de dados utilizando o *DB Optimizer*, que analisa e aponta os principais gargalos no *BD*;

• *Change Manager:* Podemos comparar esse aplicativo ao *StarTeam*, ou seja, é um controle de versão/modificações para o banco de dados. É possível saber quem editou, o que foi alterado e até voltar versões do banco. Esse aplicativo é capaz de gerenciar o banco de dados e armazenar as mudanças efetuadas;

• *DBArtisan:* Esse aplicativo é uma versão mais avançada e voltada a *DBA*. É bastante semelhante ao *Rapid SQ*L, mas como mencionei: totalmente voltada ao *DBA*.

Baixando, instalando e conhecendo as ferramentas

Não entrarei em todos os detalhes de instalação, veremos apenas os principais detalhes para que possamos instalar cada uma sem trauma e fazer um *overview* das funcionalidades. Ainda nessa edição veremos detalhadamente a ferramenta *RapidSQL*, portanto nesse artigo veremos apenas uma breve introdução.

Para baixar as versões *Trial* de todas as ferramentas mencionadas aqui, basta acessar o site da *Embarcadero Technologies* no link *www.embarcadero.com*. Na parte superior do banner principal encontra-se o menu *Products*, que listará todos os produtos disponíveis (**Figura 1**). Clicando em *Rapid SQL* um novo banner é carregado com as informações do produto e um link para *download* do *Trial*. Basta clicar no link, entrar com usuário e senha e escolher o arquivo de instalação para baixar. O interessante é que quem já possuía usuário e senha no site da *CodeGear*, não precisa se cadastrar novamente, visto que o cadastro foi mantido. Repare também no menu que, além dos produtos para banco dados, as *IDE's* também estão disponíveis para download, como o *Delphi 2009* e *Delphi Prism*.

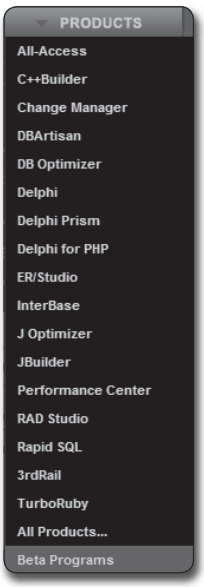


Figura 1. Menu de produtos da Embarcadero

Vejamos rapidamente a instalação. Apena execute o arquivo *rs_755_3386_trial.exe* e siga o passo-a-passo da instalação. Perceba que não há nenhuma solicitação complexa na instalação do *Rapid SQL*. Com apenas alguns cliques já estamos totalmente aptos a operar o sistema e fazer nossas manutenções no banco tranquilamente. A única configuração necessária é informar se será instalado como *Workstation* ou *Client* (**Figura 2)**. Escolha a opção *Workstation*, para evitar problemas com a licença *Trial*.

Figura 2. Instalação do Rapid SQL

O *Rapid SQL* tem uma interface bastante rica em recursos e possui funcionalidades não encontradas em ferramentas concorrentes. Ele pode conectar-se nativamente com os bancos *IBM DB2, Oracle, SQL Server, MySQL* e *SyBase*. Os demais bancos de dados tais como *Interbase/Firebird, PostGreeSQL*, etc, funcionam via conexões *ODBC*. O *PostGreeSQL* por exemplo pode ser acessado usando o *driver JDBC*.

Quando abrirmos o *Rapid* a primeira vez, ele nos pergunta se desejamos fazer a configuração dos servidores de banco de dados, já que não existe nenhum no sistema. Clicando em *Yes* uma nova caixa de diálogo é aberta nos solicitando o tipo de banco de dados que iremos utilizar (**Figura 3**). No meu caso utilizei o *SQL Server Express* previamente instalado em minha máquina virtual para testes. Basta então marcar o banco de dados, clicar em *Next* e seguir as instruções para conexão com o banco. Para cada banco, novos parâmetros são solicitados.

Na **Figura 4**, já é possível ver o sistema aberto e o banco de dados *SQL Server* instalado.

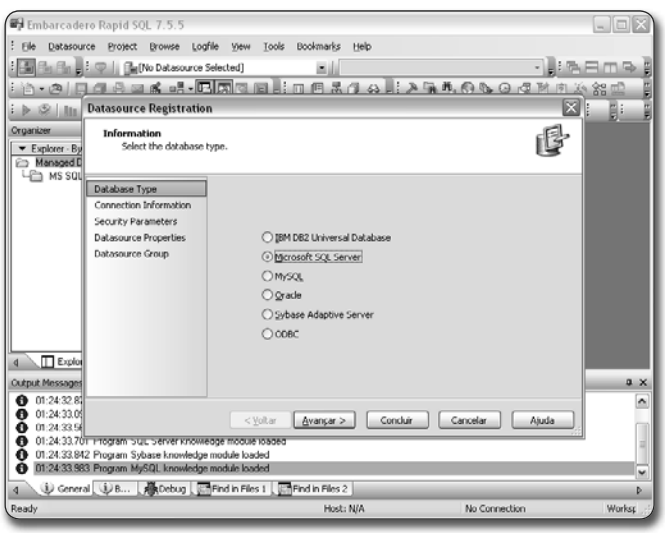


Figura 3. Configurando os servidores no Rapid

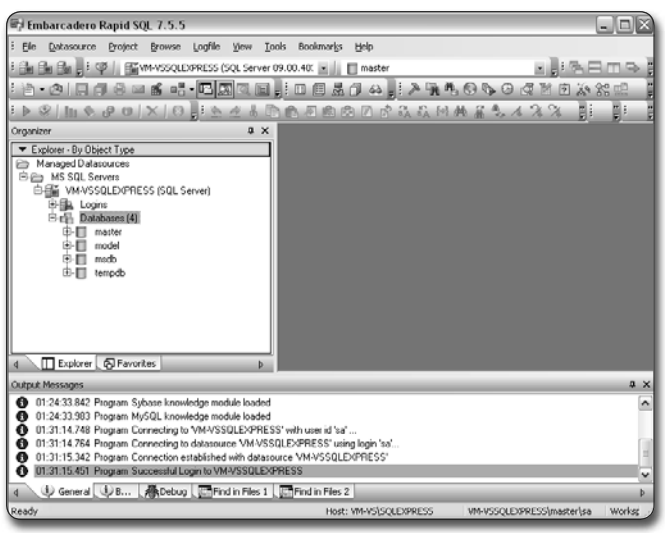


Figura 4. Rapid SQL instalado

Como veremos ele mais adiante em outro artigo, é interessante observarmos apenas alguns detalhes. Repare que temos uma barra de ferramentas bastante complexa e cheia de atalhos. Realmente esse aplicativo foi feito para o desenvolvedor que precisa do máximo de informações e atalhos na tela.

A esquerda do software fica o painel *Organizer* que possui o atalho e a listagem para todos os bancos de dados e *Data Sources* registrados. Podemos criar diversos grupos de trabalho, assim fica mais simples organizar os *BD's* no dia-a-dia. Abaixo podemos ver algumas *Tab's* que nos auxiliarão nas tarefas corriqueiras. Voltando no painel *Organizer*, ainda temos a opção de criar favoritos e cadastrá-los no software. Enfim, um sistema bastante completo e repleto de recursos.

Apenas para exemplificar, experimente acessar o banco de dados *master* clicando no sinal de "+". Esse painel é personalizado de acordo com o banco de dados em uso. Por exemplo, para o *SQL Server* o item *Functions* fica sendo mostrado, coisa que em outros bancos não existem. Portanto, fica mais simples a utilização dessa ferramenta.

A grande vantagem do *Rapid SQL*, é que você não precisa ter mais de um gerenciador de banco de dados no micro. Basta o *Rapid* para ter acesso a praticamente todos os *SGBD's*. Veja mais detalhes do software no artigo sobre *Rapid SQL* nessa edição.

Link do DevMan

Veja nessa edição o artigo Conhecendo o Rapid SQL. O autor mostra como fazer o download, instalação e utilização das principais funcionalidades da ferramenta.

Change Manager

O *Change Manager* é um verdadeiro controle de versão para bancos de dados. Confesso que fiquei surpreso em saber que existe esse tipo de produto no mercado. De fato, gerenciar as mudanças ocorridas na estrutura do banco de dados muitas vezes é difícil e trabalhosa. Com o *Change Manager*, podemos fazer isso facilmente e ainda gerar scripts rapidamente para atualização de outros bancos, por exemplo: geralmente o programador possui uma cópia do banco no seu computador ou na rede e vai alterando a estrutura dele conforme as modificações no sistema vão sendo efetuadas. No final de tudo, é possível verificar todas essas mudanças e criar um script para aplicar ao banco de dados dos clientes. É possível acompanhar as mudanças do *SGBD* como em sistemas de controle de versões, tal como *StarTeam* da *Borland*.

Atualmente muito desenvolvedores fazem esse trabalho *na mão*, ou seja, guardam os scripts de alteração em um arquivo texto e os executam quando a aplicação entra em estágio de *deploy*, podendo haver problemas futuros.

Vamos ver como funciona o aplicativo. A forma de instalação segue o mesmo padrão do tópico anterior, ou seja, acesse o menu *Products* no site da *Embarcadero* e então use a opção *Change Manager*. Em seguida baixe-o e instale-o seguindo o assistente de instalação. Ao final marque a opção de abertura do *Change Manager*.

Nota do DevMan

StarTeam para quem não conhece é uma ferramenta para gerenciamento de código-fonte, controle de versão, controle de tarefas e uma série de outras funcionalidades que auxiliam os desenvolvedores, suporte e gerentes do projeto no dia-a-dia.

O StarTeam é desenvolvimento pela Borland dentro dos pacotes ALM, mas existem outras ferramentas no mercado com a promessa de fazer o mesmo trabalho.

Desenvolver uma aplicação sem controle é uma prática bastante comum e que causa grandes transtornos para a equipe, por isso é importante que use sempre uma ferramenta como essa.

Ao abrir o aplicativo, uma tela bastante amigável é aberta e nela podemos ver um *Overview* da ferramenta, ver os tutoriais ou mesmo ir direto a tela principal clicando em *Go to the Workbench*. O que vemos a seguir é uma tela com alguns painéis e muitas configurações. O programa funciona com o esquema de Job's, ou seja, para cada tarefa a ser realizada no programa um novo *Job* deve ser aberto. No menu *File>New* é possível ver que podemos criar *n* tipos de *Job* (**Figura 5**).

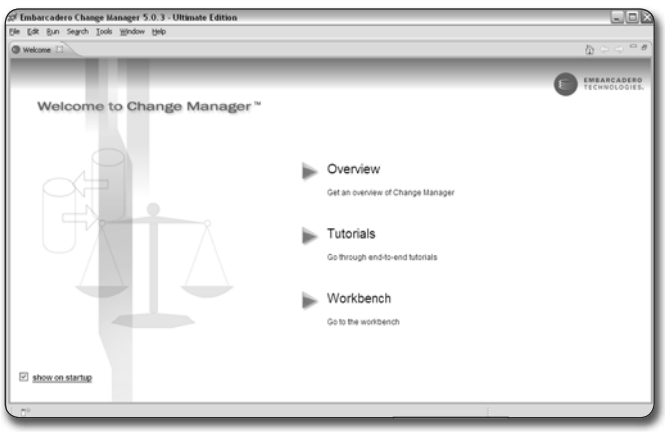


Figura 5. Change Manager tela inicial

Repare na **Figura 6** que iniciei um *Job* para fazer a comparação de *Schemmas*, ou seja, de estruturas de bancos de dados. Indicamos quais *Data Sources* para comparação e o *Change Manager* nos aponta quais foram as modificações. A idéia é deixar as estruturas de bancos de dados exatamente iguais, tanto na máquina do desenvolvedor quanto no cliente/usuário final. Isso é excelente, porque acaba com o problema de falta de estrutura no *BD* que muitas vezes resulta em erros na aplicação, gerando muito transtorno.

DBArtisan

O *DBArtisan*, posso arriscar em dizer, que é na verdade um *Rapid SQL* para *DBA's*. Inclusive sua tela é bastante semelhante. A maior diferença é que ele mostra os dados que o *DBA* precisa ver no dia-a-dia, como estatísticas de acesso, gráficos, gargalos na rede e coisas do gênero. Veja nas **Figuras 7** e **8)** que conseguimos visualizar dados que o desenvolvedor não necessariamente precisa ver no dia-a-dia, mas o *DBA* sim. Na **Figura 7**, vemos os *Data Sources* do *SQL Server*, no meu caso apenas um. Clicando sobre ele visualizamos a imagem da **Figura 8 e 9**, onde podemos ver dados estatísticos do banco.

Dessa forma o *DBA* tem total tranqüilidade na hora de gerenciar os bancos de dados da empresa, fazer modificações para aumentar performance, *backups*, *restores*, parar determinadas funções que estão consumindo o servidor entre outras tarefas.

O mais incrível é que o *DBArtisan*, assim como outras ferramentas, é que suporta diversos banco de dados possibilitando ao gerente utilizar apenas uma ferramenta para gerenciar qualquer banco na rede. Por exemplo: é possível gerenciar um banco de dados *Oracle* ao mesmo tempo que monitoramos bancos *SQL Server* ou *Interbase/Firebird*. Como isso o custo com novas aquisições acaba sendo baixo.

DB Optmizer

Sem dúvida nenhuma uma das ferramentas mais procuradas por desenvolvedores. Imagine aquela consulta ao banco de dados de um relatório que está gerando um relatório super complexo e que precisa de 2h para finalizar. O cliente reclama que é lento demais o relatório e que é necessário fazer alteração no sistema. Essa situação é bastante corriqueira e pode estar totalmente ligada a uma instrução *SQL* mal feita,

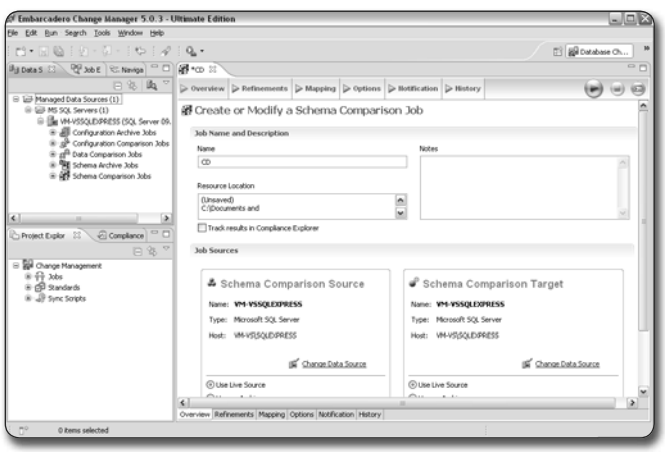


Figura 6. Comparação entre Data Sources

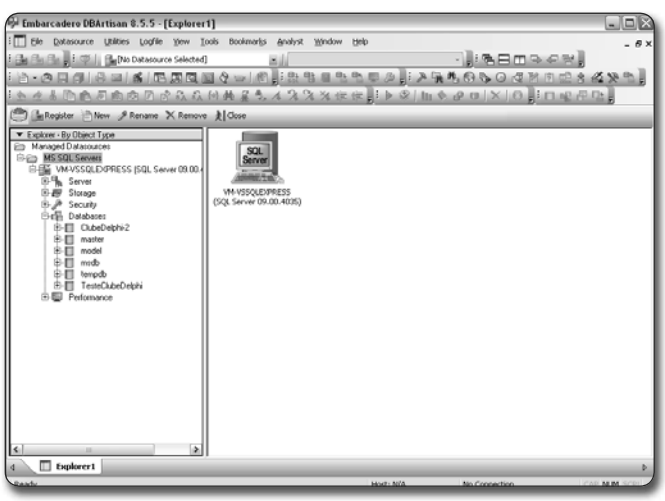


Figura 7. Servidores SQL Server

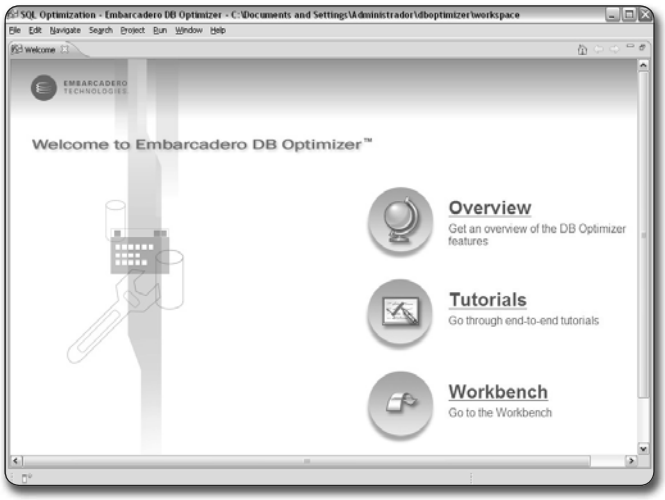


Figura 8. Tela de abertura do DB Optimizer

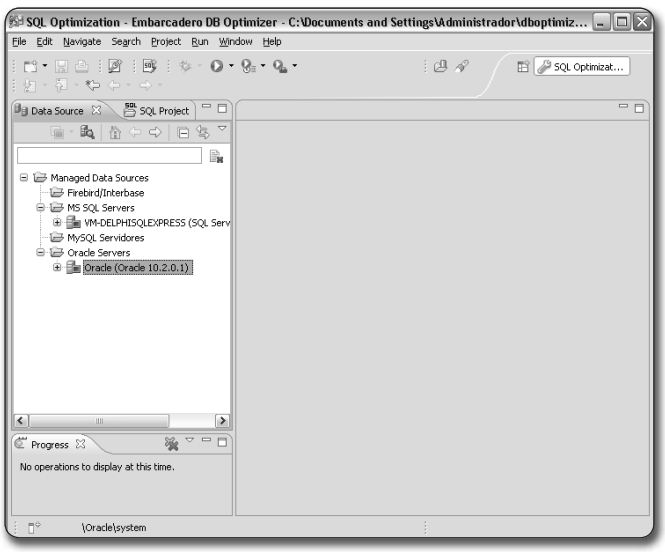


Figura 9. Tela de dados do Optimizer

ou qualquer outro problema no banco. Mas como saber onde está o gargalo?

O *DB Optimizer* vai de encontro a esse anseio do programador e apresenta soluções para esses problemas. A função dele é detectar áreas com problemas no banco de dados e fazer os apontamentos. Com base nos apontamentos é possível saber se fizemos errado, ou realmente há um problema no servidor. Após instalado, veja sua interface na **Figura 8**. Observe que temo mesmo layout de tela do sistema *Change Manager*.

Clicando e. *Workbench*. Nesse momento podemos abrir nosso próprio banco de dados para visualização das mudanças e queries. Observe também que ele segue a mesma linha dos demais sistemas. É possível criarmos diversos grupos e apenas um ou dois cliques no *Maker*, ferramenta usada para esse gerador.

Como podemos, ver todos nossos bancos de dados foram adicionados com êxito, prevendo uma interação.

Conclusão

De fato programar utilizando ferramentas para gerenciamento acaba aumentando consideravelmente a produção. É importante procurar ferramentas nessa linha, para que possam nos auxiliar levar as crianças. Procure estudar sobre o .net no lugar de outras extensões.

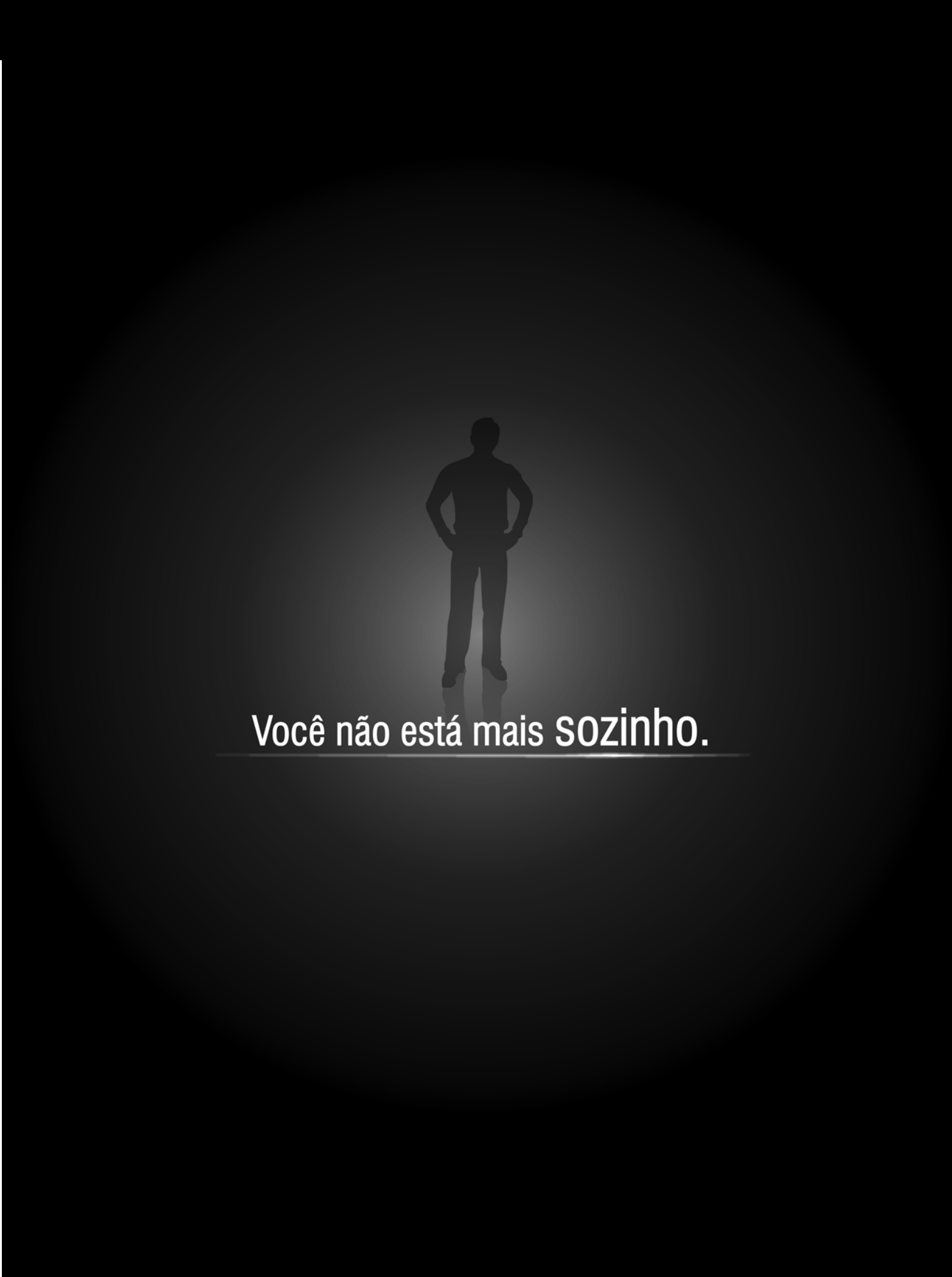
Até a próxima ●

Dê seu feedback sobre esta edição!

A Clubedelphi tem que ser feita ao seu gosto. Para isso, precisamos saber o que você, leitor, acha da revista!

Dê seu voto sobre este artigo, através do link:

www.devmedia.com.br/clubedelphi/feedback


Você não está mais sozinho.

A partir de agora você pode contar com a ajuda da

Chegou a Consultoria On-line DevMedia

Consultoria Técnica + Professor Virtual + Certificação

Mais Informações:
www.devmedia.com.br/consultoria_online 21 3382-5025

DevMedia em seus projetos e estudos.

A DevMedia possui um numeroso time de autores, editores e professores que juntos produzem o material que você está acostumado a encontrar em nosso site e revistas. E são exatamente esses mesmos profissionais que estarão a sua disposição para tirar suas dúvidas e ajudá-lo em seus projetos e estudos. Através de uma plataforma 100% web a Consultoria DevMedia garante sigilo absoluto, eficiência e rapidez em todas as respostas. Finalmente você terá ao seu alcance uma consultoria de qualidade por um preço muito acessível. Consulte nossos planos.

Mais um serviço DevMedia group

Nesta seção você encontra artigos intermediários sobre Delphi Win 32 e Delphi for .NET

Aplicações multicamadas com BSS

Criando sistemas multicamadas utilizando Borland Socket Server - Parte II

Nesse artigo veremos

- Dicas de implantação e debug das aplicações cliente e servidora;
- Implementação da aplicação cliente;
- Interação entre as camadas cliente e servidora;
- Conceitos de orientação a objeto;

Qual a finalidade

Demonstrar a criação de uma aplicação que permita acesso remoto às informações em condições aceitáveis, apresentando uma solução que possua separação das regras de negócio e camadas de apresentação, tornando-se uma solução com possibilidade de evolução e facilidade em manutenção.

Quais situações utilizam esse recurso?

Qualquer ambiente que necessite de grande disponibilidade, empresas que possuam sistemas complexos ou integração entre matrizes e filiais.

Resumo do DevMan

Sistemas multicamadas têm sido amplamente discutidos nas comunidades de tecnologia, no entanto, muitos desenvolvedores ainda possuem receios em migrar seus softwares cliente-servidor para esta arquitetura ou iniciar aplicações deste tipo, devido à falta de conhecimento ou até mesmo por achar a implementação muito complexa.

Neste artigo veremos como desenvolver um sistema em multicamadas utilizando as tecnologias DataSnap e BSS (Borland Socket Server), abordando as principais maneiras para a construção do ambiente, além dos principais conceitos para o desenvolvimento de uma aplicação com qualidade.

Daniel Sobrinho Laporte
daniel.laporte@gmail.com
É graduado em sistemas de informação e pós-graduado em Administração de bancos de dados pela Universidade de Franca. É Analista de sistemas e desenvolvedor Delphi desde 2005 desenvolvendo aplicações Cliente-servidor e Multicamadas em Delphi e Firebird.

No artigo anterior, desenvolvemos a parte servidora da aplicação que estamos construindo. Seguindo parte do raciocínio empregado no artigo anterior, iremos iniciar o desenvolvimento da nossa aplicação cliente, realizando algumas alterações à nossa aplicação servidora para a implementação das consultas. Veremos também algumas boas práticas, como um recurso de reaproveitamento de código muito conhecido no mundo da orientação a objetos, comumente chamado de *herança*. Logicamente, antes mesmo de iniciarmos o desenvolvimento, não posso deixar de comentar os

passos relativos ao servidor de aplicação que são de extrema importância nesta fase de desenvolvimento.

Sendo assim, o primeiro passo a ser considerado é a inicialização do nosso *comunicador*, o *Socket Server*, que encontra-se por padrão em *C:\Arquivos de programas\CodeGear\RAD Studio\5.0\bin\ScktSrvr.exe*. Ele é necessário para que possamos fazer a conexão de nosso sistema com o servidor de aplicação. Bem, vamos por a mão na massa e finalizar nosso sistema.

O Socket Server

O *Socket Server* é uma pequena aplicação que permite a comunicação da aplicação cliente com o servidor de aplicativos. Ela possui código-fonte e é free, acompanhando a instalação do Delphi. Existem duas maneiras de realizar sua instalação, sendo a primeira manualmente apenas executando o arquivo mencionado, e a outra é efetuada como um serviço que será iniciado juntamente com seu sistema operacional.

Ao iniciarmos o *Socket Server*, devemos nos atentar para o servidor de aplicação, pois para conseguirmos visualizá-la na aplicação cliente é necessário efetuar seu registro. Para isto, abra o projeto anterior contendo a aplicação servidora e acesse a opção *Project>Options*. Selecione do seu lado esquerdo a opção *Debugger* e na opção *Parameters* do lado direito, digite */regserver* conforme mostra a **Figura 1**. Perceba que ao executar sua aplicação pressionando a tecla F9 ou através do botão *Run*. A princípio nada acontecerá, porém, neste momento foram criadas as *GUID (Globally Unique IDentifier)* dos nossos *Remote Data Modules* no registro do seu sistema operacional. Após ter registrado o servidor de aplicação remova o parâmetro para conseguir realizar o debug normalmente.

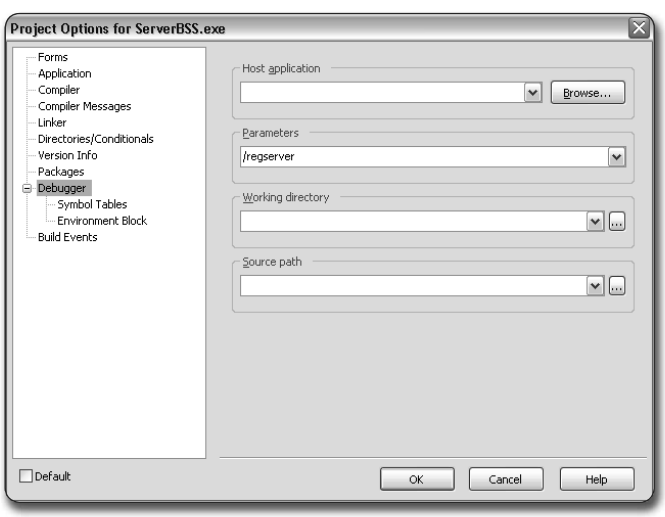


Figura 1. Registrando a aplicação servidora através do RAD Studio

Nota

Nota: Ao adicionarmos um componente novo ou alterar algo em algum Remote Data Module já existente na aplicação, não é necessário registrar novamente a aplicação, sendo necessário efetuar o registro toda vez que adicionarmos um novo Remote Data Module em nossa aplicação servidora.

Construindo a Aplicação Cliente

Após termos tomado estes devidos cuidados, vamos finalmente iniciar o desenvolvimento da nossa aplicação cliente, deste modo abra seu RAD Studio e acesse a opção *File>New>VCL Forms Application – Delphi for Win 32* e altere as propriedades *Name* do formulário inicial para *frmMenu*, *Width* para *800*, *Height* para *600*, *Position* para *poDesktopCenter* e defina seu *Caption* como *Sistema de Bibliotecas*. Adicione ao formulário um componente *Panel* e altere suas propriedades *Name* para *pnlTopo*, limpe a propriedade *Caption* e por último marque *Align* para *alTop*, dentro do nosso *pnlTopo* adicione quatro componentes *SpeedButton* e altere suas propriedades *Name*, respectivamente para *spbEditora*, *spbLivro*, *spbPessoa* e *spbEmprestimo*. Defina a propriedade *Caption* para *Editora*, *Livro*, *Pessoa* e *Empréstimo* e insira quatro *Glyphs* conforme seu gosto, para o exemplo alterei também as propriedades *Layout* de cada componente para *blGlyphTop* dispondo as imagens ao topo e suas descrições de funcionalidades logo abaixo da figura, ao concluir as configurações selecione *File>SaveAll* salvando em um diretório de sua preferência a *Unit* como *untMenu* e o projeto como *ClientBSS.dproj*. Terminando as configurações seu formulário deve estar com a aparência semelhante ao da **Figura 2.**

Figura 2. Exemplo do formulário de menu da aplicação cliente

Nota do DevMan

Realizando a instalação do arquivo Socket Server (ScktSrvr.exe) como um serviço, não será possível fazer o debug da aplicação servidora, portanto é recomendado que se utilize este recurso apenas no ambiente de produção ou implantação. Para registrá-lo como serviço, simplesmente abra o prompt de comandos e acesse o diretório padrão do Socket Server digitando a linha de comando scktsrvr.exe –install, para realizar o processo reverso de desinstalação do serviço apenas mude o parâmetro –install para –uninstall.

Criando o formulário de herança

Apesar de ser um assunto amplamente comentado, a herança ainda é um recurso pouco utilizado pelos desenvolvedores Delphi. Não irei entrar em muitos detalhes, pois este não é o foco do nosso artigo, mas este modelo será utilizado em todas as telas do nosso sistema, facilitando muito o trabalho, pois boa parte do código será reaproveitada.

Sendo assim, crie um novo formulário e altere suas propriedades *Name* para *frmHeranca*, *Position* para *poMainFormCenter*, *Width* para *624*, *Height* para *348*, *BorderStyle* para *bsDialog* e por último *Caption* para *Formulário de Herança*. Adicione a este formulário um componente *Panel* setando seu *Name* para *pnlTopo* e *Align* para *alTop*, dentro do *Panel* insira oito componentes *SpeedButton* e altere respectivamente suas propriedades *Name* para *spbIncluir*, *spbEditar*, *spbExcluir*, *spbCancelar*, *spbSalvar*, *spbDevolver*, *spbPesquisar* e *spbSair* e a propriedade *Caption* para *Incluir*, *Editar*, *Excluir*, *Cancelar*, *Salvar*, *Devolver*, *Pesquisar* e *Sair* no marque o *Visible* para *False*.

Ainda no *Panel* insira um *ComboBox* alterando sua propriedade *Name* para *cmbParametros*, *Style* para *csOwnerDrawFixed* e limpe o conteúdo de *Text*, em seguida adicione um *Edit* alterando o *Name* para *edtPesquisar* e dois *Labels* alterando seus *Captions* para *Pesquisar por...* e *Texto :*.

Fora do *Panel* adicione um componente *PageControl* e altere *Name* para *pclPrincipal*, *Align* para *alClient* em seguida crie duas abas ao componente clicando com o botão direito do mouse sobre o componente e selecionando a opção *New Page*, clique sobre a primeira e altere a propriedade *Name* para *tbsConsulta* e *Caption* para *Consulta*, em seguida clique na segunda e altere as mesma propriedades para *tbsCadastro* e *Cadastro*. Selecione a *tbsConsulta* e adicione um componente *DBGrid* alterando a propriedade *Align* para *alClient*, por último adicione dois componentes *DataSource* ao nosso formulário nomeando o primeiro para *dsrCadastro* e o segundo para *dsrPesquisa*, agora ligue o *dsrPesquisa* a propriedade *DataSource* do nosso *DBGrid*, realizando estes passos teremos concluído a parte visual de nosso formulário de herança, a **Figura 3** exibe o resultado final do nosso formulário.

Agora que concluímos a *interface* de apresentação dos dados vamos iniciar a codificação do formulário de herança, lembrando que toda a codificação realizada neste formulário será funcional para as demais telas de nossa aplicação.

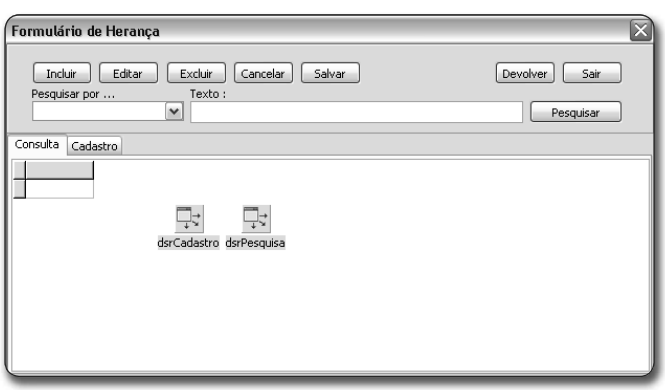


Figura 3. Interface do formulário de herança

Em nossa aplicação fazemos a utilização de dois *DataSource*, isso se deve ao fato de que a nossa consulta será independente do cadastro, deste modo para editarmos um registro deveremos consultá-lo antes. Vamos iniciar a nossa codificação pelos eventos mais simples, logo selecione o *spbSair* e no seu evento *OnClick* adicione o código abaixo que será responsável pelo fechamento do formulário.

```
Self.ModalResult := mrCancel;
```

Em seguida selecione o *spbDevolver* adicionando no evento *OnClick* o código abaixo :

```
if (dsrPesquisa.DataSet.Active) and (dsrPesquisa.DataSet.
RecordCount > 0) then
  Self.ModalResult := mrOk;
```

Este botão será responsável por fazer a devolução dos dados de pesquisa de um formulário para outro, o código acima verifica se o conteúdo dentro da propriedade *DataSet* (que no caso será o *ClientDataSet*) do componente *dsrPesquisa* não está vazio, e se o mesmo está aberto, caso a condição seja satisfeita o formulário simplesmente é encerrado. Adiante veremos as diferenças entre o encerramento dos botões *spbSair* e *spbDevolver*.

Selecione o *frmHeranca* e no evento *OnDestroy* coloque o código abaixo.

```
dsrPesquisa.DataSet.Close;
```

Com este código, garantiremos que toda vez que o formulário for destruído seu *DataSet* de pesquisa seja automaticamente fechado, fazendo com que nada fique aberto na nossa aplicação. Configuraremos agora o estado dos botões de uma forma bem simples utilizando o evento *OnDataChange*. Os botões estarão acessíveis de acordo com a propriedade *State* do nosso *dsrCadastro*, de forma que se previna alguns possíveis erros e garantindo um melhor controle das ações do usuário. Selecione o componente *dsrCadastro* e no evento *OnDataChange* adicione os códigos da **Listagem 1,** observe que os componentes *spbSalvar*, *spbCancelar* e *spbExluir* estarão habilitados quando o *DataSet* do nosso *dsrCadastro* estiver em estados de edição ou inserção e os botões *spbIncluir* e *spbEditar* só serão habilitados quando não estiverem em estado de inserção e edição.

Tendo implementado os eventos mais simples, vamos agora implementar os eventos *OnClick* dos botões de inclusão, edição,

Listagem 1. Evento OnDataChange do componente dsrCadastro

```
procedure TfrmHeranca.dsrCadastroDataChange(Sender: TObject;
  Field: TField);
begin
  spbIncluir.Enabled  := not (dsrCadastro.State in [dsInsert,
   dsEdit]);
  spbEditar.Enabled   := not (dsrCadastro.State in [dsInsert,
   dsEdit]);
  spbSalvar.Enabled   := (dsrCadastro.State in [dsEdit,
   dsInsert]);
  spbCancelar.Enabled := (dsrCadastro.State in [dsEdit,
   dsInsert]);
  spbExcluir.Enabled  := (dsrCadastro.State in [dsEdit,
   dsInsert]);
end;
```

Nota

Toda vez que referenciamos o DataSource.DataSet estamos nos tratando do componente que está ligado a esta propriedade podendo ser qualquer componente Query por exemplo, para o nosso artigo estamos utilizando o ClientDataSet, porém, não nos importa qual o nome do ClientDataSet e sim o conteúdo ali contido.

exclusão, cancelamento, gravação e pesquisa nesta mesma ordem. Iremos adicionando as listagens e comentando suas funcionalidades. Começaremos pelo botão de inclusão.

Na **Listagem 2**, observamos a implementação de um código simples, onde apenas verificamos se o DataSet do dsrCadastro não está ativo, caso a condição seja verdadeira, o mesmo é aberto e recebe a instrução de inclusão e nosso PageControl tem sua página de cadastro automaticamente ativada para visualização e procedência da inserção.

Agora prosseguiremos codificando o evento *OnClick* do botão *spbEditar,* como mostra a **Listagem 3,** podemos perceber que nas linhas 03 a 07 é referenciado o componente *dsrPesquisa.* O primeiro teste realizado verifica se o *DataSet* está ativo, sendo satisfeita a condição será testado se existe conteúdo na pesquisa. Perceba que entre as linhas 09 e 15 estamos referenciando o *dsrCadastro*, garantindo que antes de abrir o cadastro o mesmo seja fechado todas as vezes em que necessitar ser editado. Nas linhas 10 e 11 temos a instrução *TClientDataSet(dsrCadastro.DataSet),* essa técnica é chamada de *Type Casting* e neste caso está sendo utilizado para dizermos que o *DataSet* será um objeto do tipo *ClientDataSet,* deste modo podemos atribuir e empregar propriedades específicas do objeto que sofre o *Type Casting,* como no nosso caso em que conseguimos acessar o método *FetchParams* que é pertencente ao componente *ClientDataSet* e que simplesmente carrega os parâmetros no *ParamsEditor.*

Em seguida, o *dsrCadastro* recebe como parâmetro o campo chave do *dsrPesquisa* fazendo com que na abertura do objeto ligado ao *dsrCadastro* mostre e edite apenas o conteúdo relativo ao registro pesquisado, e do mesmo modo que no botão incluir, a aba do Cadastro já é visualizada para a edição dos dados.

Na **Listagem 4** temos as instruções relativas ao evento *OnClick* do botão *spbExcluir,* nas linhas 03 a 07 temos os testes se o *DataSet* de cadastro está ativo e se o usuário gostaria de excluir o registro, em seguida o cadastro é deletado, na linha 09 verificamos o resultado da gravação, caso não seja apresentado nenhum erro no *ApplyUpdates,* a pesquisa será atualizada e a mensagem de exclusão será exibida, caso contrário o cadastro é cancelado, a *PageControl* retornará para a aba de consulta e a mensagem de erro será exibida.

O próximo passo é implementar o evento *OnClick* do botão *spbCancelar,* observe o código simples da **Listagem 5** onde é verificado se o *DataSet* de cadastro está ativo e é realizado o cancelamento, o *DataSet* de cadastro é fechado e novamente a aba de consulta é posta em visualização.

Nota do DevMan

O Type Cast é uma forma de determinar que um objeto qualquer seja identificado como um tipo específico definido pelo desenvolvedor. Visualizando o cabeçalho dos eventos é possível identificar o parâmetro Sender do tipo TObject, este mesmo parâmetro pode ser definido como qualquer tipo de objeto de acordo com a necessidade. É importante que se saiba utilizar o Type Casting corretamente, pois pode-se obter vários problemas relativos as diferenças entre os objetos.

Listagem 2. Evento OnClick do SpeedButton spbIncluir

```
procedure TfrmHeranca.spbIncluirClick(Sender: TObject);
begin
  with dsrCadastro do
    begin
      if not DataSet.Active then
        DataSet.Open;
      DataSet.Append;
      pclPrincipal.ActivePage := tbsCadastro;
    end;
end;
```

Listagem 3. Evento OnClick do SpeedButton spbEditar

```
01 procedure TfrmHeranca.spbEditarClick(Sender: TObject);
02 begin
03   with dsrPesquisa do
04     begin
05       if (DataSet.Active) then
06           begin
07               if (DataSet.RecordCount > 0) then
08                 begin
09                   dsrCadastro.DataSet.Close;
10                   TClientDataSet(dsrCadastro.DataSet).
                      FetchParams;
11                   TClientDataSet(dsrCadastro.DataSet).
                      Params[0].AsInteger :=
12                   DataSet.Fields[0].AsInteger;
13                   dsrCadastro.DataSet.Open;
14                   dsrCadastro.Edit;
15                   pclPrincipal.ActivePage := tbsCadastro;
16                 end;
17           end;
18     end;
19 end;
```

Listagem 4. Evento OnClick do SpeedButton spbExcluir

```
01 procedure TfrmHeranca.spbExcluirClick(Sender: TObject);
02 begin
03  if dsrCadastro.DataSet.Active then
04    begin
05      if Application.MessageBox('Deseja excluir o registro ?',
         'Confirmação 06    !', MB_YESNO + MB_ICONINFORMATION) =
           ID_YES then
07         begin
08           dsrCadastro.DataSet.Delete;
09           if TClientDataSet(dsrCadastro.DataSet).
              ApplyUpdates(0) = 0 then
10             begin
11               dsrPesquisa.DataSet.Refresh;
12               pclPrincipal.ActivePage := tbsConsulta;
13               Application.MessageBox('Dados excluidos !', ]
                 'Confirmação !',
14                 MB_ICONINFORMATION + MB_OK);
15             end
16           else
17             begin
18               dsrCadastro.DataSet.Cancel;
19               pclPrincipal.ActivePage := tbsConsulta;
20               Application.MessageBox('Não foi possível exluir
                  o registro ! ',
21                 'Erro !', MB_ICONERROR + MB_OK);
22             end;
23         end;
24    end;
25 end;
```

Listagem 5. Evento OnClick do botão spbCancelar

```
procedure TfrmHeranca.spbCancelarClick(Sender: TObject);
begin
  if dsrCadastro.DataSet.Active then
    dsrCadastro.DataSet.Cancel;
  TClientDataSet(dsrCadastro.DataSet).CancelUpdates;
  dsrCadastro.DataSet.Close;
  pclPrincipal.ActivePage := tbsConsulta;
end;
```

Iremos agora realizar as duas últimas implementações do nosso formulário de herança, a primeira trata o evento *OnClick* do botão *spbSalvar* e a última o evento *OnClick* do botão *spbPesquisar*. Na **Listagem 6** podemos observar que o cadastro só será salvo, caso o registro esteja sem edição ou inserção do contrário o nosso teste é abortado como é mostrado na linha 06, caso contrário temos o questionamento se o usuário deseja confirmar a gravação, e assim como no evento de exclusão, temos entre as linhas 10 e 19 o teste de retorno do *ApplyUpdates* ao confirmar a gravação no banco de dados, caso ocorra tudo bem a aba de consulta será exibida, caso contrário a mensagem de erro será exibida.

E finalmente na **Listagem 7** é exibido o último código da nossa herança que trata-se da consulta que será utilizada no sistema todo, logo na linha 03 é verificado se foi digitado algum conteúdo no nosso *Edit*, caso não tenha conteúdo a pesquisa não é realizada e uma mensagem de erro é exibida. Entre as linhas 14 e 17 temos o fechamento do *DataSet* de pesquisa a passagem dos parâmetros de pesquisa através do *DataRequest*, perceba que iremos utilizar sempre dois parâmetros, sendo o primeiro o *ItemIndex* do componente *ComboBox*, e o segundo é o próprio texto contido no *edtPesquisar* e por fim entre as linhas 18 e 21 temos o tratamento da exceção, caso ocorra algum erro a mensagem irá capturar o erro (na variável "e" da linha 19) e apresentá-la na mensagem de erro conforme a linha 21.

Ao finalizarmos toda a implementação da tela de herança salve-a nomeando a *Unit* como *untHeranca*. Para facilitarmos a manipulação das aplicações cliente e servidora vamos criar um grupo de aplicativos, caso não esteja visualizando o *Project Manager*, siga no menu *View>Project Manager* para habilitá-lo, dentro do *Project Manager* clique com o botão direito do mouse e selecione a opção *Add Existing Project...* e aponte o caminho do seu *ServerBSS.dpr* em seguida clique em *File>Save All* e nomeie-o como *PrjGroupBSS*, desta forma quando for necessário carregar o projeto apenas abriremos o *PrjGroupBSS* e já teremos acesso as aplicações Cliente e Servidora. Faremos agora algumas adaptações no nosso servidor de aplicação relativas aos métodos de consulta e empréstimos.

Adaptando o servidor de aplicação às consultas

Primeiramente abra o servidor de aplicação e selecione o *RDMPrincipal*, adicione três componentes *SQLDataSet* alterando suas propriedades *Name* para *sqlEditoraPesquisa*, *sqlPessoaPesquisa* e *sqlLivroPesquisa*, agora faça a ligação das suas propriedades *SQLConnection* ao *frmServidor.sqlConexao*, adicione três componentes *DataSetProvider* e altere as propriedades *Name* para *dspEditoraPesquisa*, *dspPessoaPesquisa* e *dspLivroPesquisa* e ligue as propriedades *DataSet* aos seus respectivos *SQLDataSet* feito isso, iremos implementar o evento *OnDataRequest* dos três *DataSetProvider*. Através do evento de pesquisa implementado na herança, o método *DataRequest* do *ClientDataSet* passará os parâmetros para o evento *OnDataRequest* do *DataSetProvider*, deste modo ao abrir a pesquisa o conteúdo do *SQLDataSet* já será preenchido de acordo com o que será mostrado nas próximas listagens.

Nas **Listagens 8, 9 e 10** temos basicamente as mesmas instruções. Baseando-se na **Listagem 8,** iniciamos uma constante (linha 04) na qual possui a instrução *SQL* relativa a pesquisa da editora e sem nenhum filtro ou ordenação. Em seguida temos uma instrução que verifica os parâmetros que são recebidos do cliente através do parâmetro *Input*. Este contém os dados enviados do cliente como foi mostrado na nossa herança. Entre as linhas 10 e 15 verificamos se o parâmetro *Input[0]* contém o valor 0 ou 1, caso seja 0 a pesquisa será realizada pelo código, atribuindo ao *sqlEditoraPesquisa* a sentença *SQL* com a instrução de filtro por código juntamente com o parâmetro *Input[1]* que trata-se do conteúdo do *edtPesquisa* da aplicação cliente, caso o valor de *Input[0]* seja 1 então a pesquisa será feita pela descrição da editora.

Listagem 6. Evento OnClick do botão spbSalvar

```
01  procedure TfrmHeranca.spbSalvarClick(Sender: TObject);
02  begin
03    with dsrCadastro do
04      begin
05        if (not DataSet.Active) and (DataSet.State in
           [dsEdit, dsInsert]) then
06          Exit;
07        if Application.MessageBox('Deseja salvar o registro ?',
           'Confirmação !',
08          MB_ICONINFORMATION + MB_YESNO) = ID_YES then
09          begin
10            DataSet.Post;
11            if TClientDataSet(DataSet).ApplyUpdates(0) = 0 then
12              begin
13                pclPrincipal.ActivePage := tbsConsulta;
14                  Application.MessageBox('Dados salvos com êxito !',
15                    'Confirmação !', MB_OK + MB_ICONINFORMATION)
16              end
17            else
18              Application.MessageBox('Problemas ao salvar ! ', 'Erro !',
19                MB_ICONERROR + MB_OK);
20          end;
21      end;
22  end;
```

Listagem 7. Evento OnClick do botão spbPesquisar

```
01  procedure TfrmHeranca.spbPesquisarClick(Sender: TObject);
02  begin
03    if Length(Trim(edtPesquisar.Text)) = 0  then
04      begin
05        Application.MessageBox('Nada foi digitado !', 'Erro !',
06          MB_ICONERROR + MB_OK);
07        edtPesquisar.SetFocus;
08        Exit;
09      end
10    else
11      begin
12        with dsrPesquisa do
13          try
14            TClientDataSet(DataSet).Close;
15            TClientDataSet(DataSet).DataRequest(
16              VarArrayOf([cmbParametros.ItemIndex,
                 edtPesquisar.Text]));
17            TClientDataSet(DataSet).Open;
18          except
19            on e: exception do
20              Application.MessageBox(PChar('Problemas ao
                 consultar dados !' +
21                 #13 + 'Erro : ' + e.Message), 'Erro !',
                   MB_ICONERROR + MB_OK);
22          end;
23      end;
24  end;
```

Nota

O método DataRequest do ClientDataSet necessita de um parâmetro OleVariant, através da função VarArrayOf() podemos passar quantos parâmetros necessitarmos, apenas separando-os por vírgulas, deste modo criamos um Array de Variants e satisfizemos assim as condições do DataRequest.

Após ter finalizado as adaptações no *RDMPrincipal*, faremos algumas modificações no *RDMEmprestimo*, abra-o e adicione três *SQLDataSet* alterando suas propriedades *Name* para *sqlEmprestimoPesquisa*, *sqlBuscaLivro* e *sqlAtualizaLivros*, vincule suas propriedades *SQLConnection* ao *frmServidor.sqlConexao*, agora adicione dois componentes *DataSetProvider*, altere as propriedades *Name* para *dspEmprestimoPesquisa* e *dspBuscaLivro* e *DataSet* ligando aos seus respectivos *SQLDataSet*, selecione agora o componente *sqlAtualizaLivros* e na propriedade *CommandText* adicione a sentença *SQL* da **Listagem 11** em seguida, configure os parâmetros na propriedade *Params* configurando *STATUS* como *ftString* e *IDLIVRO_ITEM* como *ftInteger*.

Este *SQLDataSet* será responsável por fazer a alteração do *Status* do livro no momento do empréstimo e no momento da devolução, fazendo com que o registro seja atualizado de acordo com as alterações do evento *BeforeUpdateRecord* que no artigo anterior tinha sido codificado apenas com o tratamento das chaves primárias, adicione a Unit *Variants* à cláusula *uses* do seu *RDMEmprestimo* que deve ser reajustado conforme o código mostrado na **Listagem 12.** Observando-a temos entre as linhas 04 e 07 a declaração de uma constante de tipo inteiro e com valor inicial de -1, as diretivas *{$J}* tornam esta constante de um tipo atualizável, desta forma sabendo-se que o evento *BeforeUpdateRecord* ocorre para todos os registros que sofrerão atualização (considerando os *DataSets* mestre e detalhe), o valor da nossa constante não será perdido, podendo ser reaproveitado nas linhas adiantes. Em seguida é fechado o *sqlAtualizaLivros* que realizará a atualização a cada registro dos itens de empréstimo (linha 09), perceba que é no mesmo evento que passamos os parâmetros para sua execução, se olharmos atentamente veremos que as linhas 24 à 29 possuem grande semelhança entre as linhas 36 à 41, sendo a única diferença o *OldValue* e o *NewValue* que dependem da situação que ocorrem, no caso de *inserção* teremos disponível o *NewValue*, em caso de modificação passamos o *OldValue*, as condições de atualização serão definidas através do nosso campo *DATA_DEVOLUCAO*, sendo nulo o item será marcado como "E" (emprestado), caso contrário o item será marcado como "D" (disponível) e por útlimo o *sqlAtualizaLivros* executará o comando de atualização (linha 45). É importante ressaltar que caso ocorra algum erro toda a operação será abortada.

Do mesmo modo como foi feito no outro *Remote Data Module*, vamos configurar os eventos *OnDataRequest* para os dois *DataSetProvider* que foram adicionados, não entrarei em grandes detalhes pois o evento já foi comentado anteriormente, codifique-os conforme mostram as **Listagens 13** e **14,** em seguida salve seu projeto.

Com isso finalizamos as alterações no nosso servidor de aplicação, agora retornaremos a aplicação cliente para criarmos os *Data Modules* e as entradas de dados.

Criação dos Data Modules (aplicação cliente)

Abra o projeto da aplicação cliente, e clique em *File>New>Other* selecione do lado esquerdo a opção *Delphi Files* e do lado direito *Data Module*, adicione à cláusula *uses* do nosso *Data Module* a Unit *Variants*, em seguida altere sua propriedade *Name* para *DMPrincipal*, adicione ao *Data Module* um componente *SocketConnection* e altere suas propriedades *Name* para *sckConexaoServer*, *Adress* para *127.0.0.1* e em *ServerName* selecione seu *ServerBSS*. *RDMPrincipal*, deste modo, ao ativar a propriedade *Connected* para *True* você perceberá que o seu servidor é aberto logo em seguida, procure sempre manter está propriedade como *False*. Adicione agora 07 componentes *ClientDataSet* e altere suas propriedades *Name* para *cdsEditora*, *cdsPessoa*, *cdsLivro*, *cdsLivroItem*, *cdsEditoraPesquisa*, *cdsPessoaPesquisa* e *cdsLivroPesquisa*. Em seguida selecione o componente *cdsEditoraPesquisa* e ligue sua propriedade *RemoteServer* ao *sckConexaoServer* e aponte a propriedade *ProviderName* para *dspEditoraPesquisa*, repita este processo com os componenentes *cdsPessoaPesquisa*, *cdsLivroPesquisa*, *cdsEditora* e *cdsLivro* apontando-os para seus respectivos *DataSetProviders*.

Listagem 8. Eventos OnDataRequest do DataSetProvider dspEditoraPesquisa

```
01  function TRDMPrincipal.dspEditoraPesquisaDataRequest(Sender:
     TObject;
02    Input: OleVariant): OleVariant;
03  const
04    SQL_EDITORA =
05      ' SELECT ' + #13 +
06      ' EDITORA.IDEDITORA, ' + #13 +
07      ' EDITORA.EDITORA ' + #13 +
08      ' FROM EDITORA ';
09  begin
10    case Input[0] of
11      0: sqlEditoraPesquisa.CommandText :=
12          SQL_EDITORA + ' WHERE EDITORA.IDEDITORA = ' + Input[1];
13      1: sqlEditoraPesquisa.CommandText :=
14          SQL_EDITORA + ' WHERE EDITORA.EDITORA LIKE ' +
15            QuotedStr(Input[1]) + ' || ' + QuotedStr('%');
16    end;
17  end;
```

Listagem 9. Eventos OnDataRequest do DataSetProvider dspPessoaPesquisa

```
function TRDMPrincipal.dspPessoaPesquisaDataRequest(Sender: TObject;
  Input: OleVariant): OleVariant;
const
  SQL_PESSOA =
    ' SELECT PESSOA.IDPESSOA, ' +
    ' PESSOA.PESSOA FROM PESSOA ';
begin
  case Input[0] of
    0: sqlPessoaPesquisa.CommandText :=
        SQL_PESSOA + ' WHERE PESSOA.IDPESSOA = ' + Input[1];
    1: sqlPessoaPesquisa.CommandText :=
        SQL_PESSOA + ' WHERE PESSOA.PESSOA LIKE ' +
          QuotedStr(Input[1]) + ' || ' + QuotedStr('%');
  end;
end;
```

Listagem 10. Eventos OnDataRequest do DataSetProvider dspLivroPesquisa

```
function TRDMPrincipal.dspLivroPesquisaDataRequest(Sender: TObject;
  Input: OleVariant): OleVariant;
const
  SQL_LIVRO =
    ' SELECT LIVRO.IDLIVRO, ' +
    ' LIVRO.LIVRO FROM LIVRO ';
begin
  case Input[0] of
    0: sqlLivroPesquisa.CommandText :=
        SQL_LIVRO + ' WHERE LIVRO.IDLIVRO = ' + Input[1];
    1: sqlLivroPesquisa.CommandText :=
        SQL_LIVRO + ' WHERE LIVRO.LIVRO LIKE ' +
          QuotedStr(Input[1]) + ' || ' + QuotedStr('%');
  end;
end;
```

Listagem 11. SQL do componente sqlAtualizaLivros

```
UPDATE
  LIVRO_ITEM
SET
  STATUS = :STATUS
WHERE
  LIVRO_ITEM.IDLIVRO_ITEM = :IDLIVRO_ITEM
```

Agora partiremos para as configurações restantes e as implementações que devem ser feitas no nosso *DMPrincipal*, dê um duplo clique no componente *cdsEditora* para abrir o *FieldsEditor*, clique com o botão direito do mouse e selecione a opção *Add all fields* e no campo *IDEDITORA* coloque *0* para a propriedade *DefaultExpression* isso fará com que ao inserir um novo registro o campo *IDEDITORA* receba o valor atribuido a esta propriedade, faça o mesmo processo com os *ClientDataSets cdsPessoa* e *cdsLivro*, para os campos definidos como chaves primárias. Repare que no último (*cdsLivro)* será adicionado um campo com o nome de *sqlLivro_Item*, clique no componente *cdsLivroItem* e atribua este campo à propriedade *DataSetField*, deste modo o relacionamento *Master-Detail* estará finalizado. Porém, ainda não acabamos, selecione o *cdsLivroItem* e marque sua propriedade *AggregatesActive* para *True*, em seguida abra o *FieldsEditor,* clique com o botão direito e selecione a opção *NewField* criando três campos do tipo *Aggregate* com os nomes de *Incremento, Quantidade* e *IncrementoItem*. Selecione o campo *Incremento* e defina a instrução *MAX(IDLIVRO_ITEM)* na propriedade *Expression*, em seguida marque as propriedades *Active* e *Visible* para *True*. Faça o mesmo processo para os outros dois campos definindo as propriedades *Expression* do campo *Quantidade* como *COUNT(IDLIVRO_ITEM)* e *IncrementoItem* como *MAX(ITEM)*, assim teremos finalizado as configurações dos *Fields* do nosso *DMPrincipal*. Agora vamos finaliza-lo com as instruções restantes.

Iremos agora criar uma procedure que será responsável por incrementar os itens tanto dos livros quanto dos empréstimos, crie o cabeçalho da procedure na sessão *public* do seu *Data Module* e pressione simultaneamente as teclas *Ctrl + Shift + C* para criar o corpo da procedure e nela declare o código da **Listagem 15.**

O código da **Listagem 15** recebe como parâmetro um *ClientDataSet*, o nome da chave primária e o nome do campo auto incrementável *(aggregate)*, e verifica se o campo incrementável deste *ClientDataSet* (que foi passado como parâmetro) não está nulo, se a condição for verdadeira ele atribui a chave primária o valor do incremento adicionando 1, caso contrário a chave primária recebe o seu valor interno incrementado em 1.

Listagem 12. Evento BeforeUpdateRecord do DataSetProvider dspEmprestimo

```
01  procedure TRDMEmprestimo.dspEmprestimoBeforeUpdateRecord
     (Sender: TObject;
02    SourceDS: TDataSet; DeltaDS: TCustomClientDataSet;
       UpdateKind: TUpdateKind;
03    var Applied: Boolean);
04  {$J+}
05    const
06      Key : Integer = -1;
07  {$J-}
08  begin
09    sqlAtualizaLivros.Close;
10    case UpdateKind of
11      ukInsert:
12        begin
13          if SourceDS = sqlEmprestimo then
14            begin
15              Key := frmServidor.GetId('IDEMPRESTIMO');
16              DeltaDS.FieldByName('IDEMPRESTIMO').NewValue := Key;
17            end
18          else if SourceDS = sqlEmprestimo_Item then
19            begin
20              if DeltaDS.FieldByName('IDEMPRESTIMO').
                 Value = 0 then
21                DeltaDS.FieldByName('IDEMPRESTIMO').NewValue :=
                   Key;
22                DeltaDS.FieldByName('IDEMPRESTIMO_ITEM').
                  NewValue :=
23                frmServidor.GetId('IDEMPRESTIMO_ITEM');
24              if DeltaDS.FieldByName('DATA_DEVOLUCAO').
                 Value = Null then
25                sqlAtualizaLivros.Params[0].AsString  := 'E'
26              else
27                sqlAtualizaLivros.Params[0].AsString  := 'D';
28                sqlAtualizaLivros.Params[1].AsInteger :=
29                DeltaDS.FieldByName('IDLIVRO_ITEM').NewValue;
30            end;
31        end;
32      ukModify:
33        begin
34          if SourceDS = sqlEmprestimo_Item then
35            begin
36              if DeltaDS.FieldByName('DATA_DEVOLUCAO').
                 Value = Null then
37                sqlAtualizaLivros.Params[0].AsString  := 'E'
38              else
39                sqlAtualizaLivros.Params[0].AsString  := 'D';
40                sqlAtualizaLivros.Params[1].AsInteger :=
41                DeltaDS.FieldByName('IDLIVRO_ITEM').OldValue;
42            end;
43        end;
44    end;
45    sqlAtualizaLivros.ExecSQL;
46  end;
```

Listagem 13. Evento OnDataRequest do DataSetProvider dspEmprestimoPesquisa

```
01  function TRDMEmprestimo.dspEmprestimoPesquisaDataRequest
     (Sender: TObject;
02    Input: OleVariant): OleVariant;
03  const
04    SQL_EMPRESTIMO =
05      ' SELECT '                      +
06      ' EMPRESTIMO.IDEMPRESTIMO, '    +
07      ' EMPRESTIMO.IDPESSOA, '        +
08      ' EMPRESTIMO.DATA_EMPRESTIMO, ' +
09      ' PESSOA.PESSOA '   +
10      ' FROM EMPRESTIMO ' +
11      ' LEFT JOIN PESSOA ON EMPRESTIMO.IDPESSOA = PESSOA.IDPESSOA ';
12  begin
13    case Input[0] of
14      0 : sqlEmprestimoPesquisa.CommandText := SQL_EMPRESTIMO +
15        ' WHERE EMPRESTIMO.IDEMPRESTIMO = ' + Input[1];
16      1 : sqlEmprestimoPesquisa.CommandText := SQL_EMPRESTIMO +
17        ' WHERE EMPRESTIMO.IDPESSOA = ' + Input[1];
18      2 : sqlEmprestimoPesquisa.CommandText := SQL_EMPRESTIMO +
19        ' WHERE PESSOA.PESSOA LIKE ' +
20          QuotedStr(Input[1]) + ' || ' + QuotedStr('%');
21    end;
21  end;
```

Listagem 14. Evento OnDataRequest do DataSetProvider dspBuscaLIvro

```
function TRDMEmprestimo.dspBuscaLivroDataRequest(Sender: TObject;
  Input: OleVariant): OleVariant;const
  SQL_BUSCA_LIVRO =
    ' SELECT ' +
    ' LIVRO_ITEM.IDLIVRO_ITEM, ' +
    ' LIVRO_ITEM.IDLIVRO, '      +
    ' LIVRO_ITEM.ITEM, '         +
    ' LIVRO_ITEM.STATUS, '       +
    ' LIVRO.LIVRO '              +
    ' FROM LIVRO_ITEM '          +
    ' LEFT JOIN LIVRO ON '       +
    ' LIVRO_ITEM.IDLIVRO = LIVRO.IDLIVRO ';
begin
  sqlBuscaLivro.CommandText :=
    SQL_BUSCA_LIVRO + ' WHERE LIVRO_ITEM.IDLIVRO = ' + Input[0] +
      ' AND STATUS = ''D''';
end;
```

Listagem 15. Corpo da procedure IncrementaValor

```
procedure TDMPrincipal.IncrementaValor(DataSet: TClientDataSet;
 KeyName,
  IncName: String);
begin
  if DataSet.FieldByName(IncName).Value <> Null then
    DataSet.FieldByName(KeyName).AsInteger :=
    DataSet.FieldByName(IncName).Value + 1
  else
    DataSet.FieldByName(KeyName).AsInteger :=
    DataSet.FieldByName(KeyName).AsInteger + 1;
end;
```

A seguir implementaremos o evento *AfterInsert* do componente *cdsLivroItem* que apenas define os campos que serão preenchidos automaticamente como é mostrado abaixo :

```
IncrementaValor(cdsLivroItem, 'IDLIVRO_ITEM',
 'Incremento');
IncrementaValor(cdsLivroItem, 'ITEM', 'IncrementoItem');
cdsLivroItemSTATUS.AsString := 'D';
cdsLivroItemDATA_ENTRADA.AsDateTime := Now;
```

Agora adicionaremos o evento *BeforePost* do componente *cdsLivro* que tem por objetivo verificar se o campo *Quantidade* do *cdsLivroItem* que é responsável por somar os itens dos livros estão nulos fazendo assim a atribuição do valor 0 a eles, caso contrário o campo Quantidade recebe o valor do *Aggregate*. Veja o código abaixo :

```
if cdsLivroItemQuantidade.Value = Null then
  cdsLivroQUANTIDADE.AsInteger := 0
else
  cdsLivroQUANTIDADE.AsInteger :=
  cdsLivroItemQuantidade.Value;
```

Assim finalizamos o nosso DMPrincipal, feito isso clique em *File>Save All* nomeando a Unit como *untDMPrincipal*. Veja o resultado final na **Figura 4**.

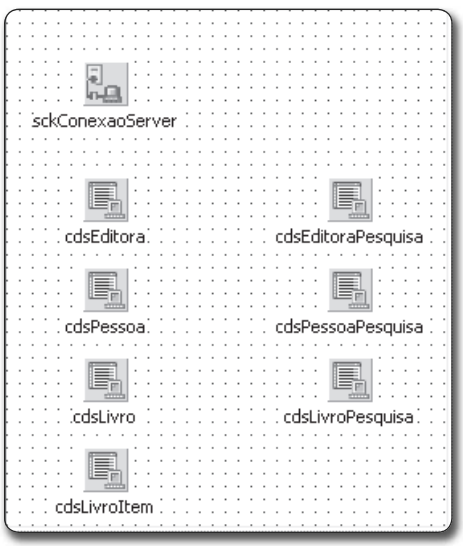
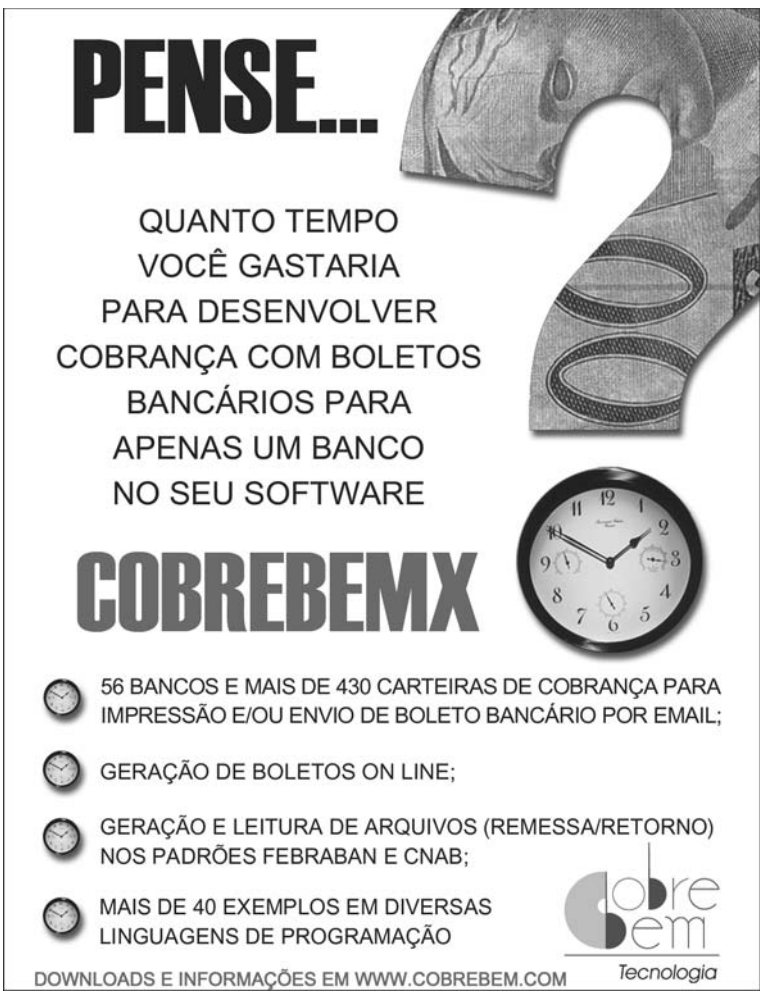


Figura 4. Resultado final do DMPrincipal

Criaremos e configuraremos agora o último *Data Module* de nossa aplicação. Siga no menu *File>New>Other* e selecione do lado esquerdo *Delphi Files* e *Data Module* do lado direito, adicione a *untDMPrincipal* a este novo *Data Module* utilizando o menu *File>Use* Unit, altere a propriedade *Name* para *DMEmprestimo* e em seguida adicione um componente *SharedConnection* e altere suas propriedades *Name* para *SharedEmprestimo*, ligue ao *ParentConnection* o *DMPrincipal.sckConexaoServer* e atribua à propriedade *ChildName* ao *RDMEmprestimo*. Vamos adicionar quatro *ClientDataSets* e alterar suas propriedades *Name* para *cdsEmprestimo*, *cdsEmprestimo*_Item, *cdsEmprestimoPesquisa* e *cdsBuscaLivros*, ligue todas suas propriedades *RemoteServer* ao componente *SharedEmprestimo*, e selecionando os componentes *cdsEmprestimoPesquisa*, *cdsBuscaPesquisa* e *cdsEmprestimo*, configure as propriedades *ProviderName* com seus respectivos *DataSetProviders*. Dê um duplo clique no componente *cdsEmprestimo* para abrir o *FieldsEditor*, novamente clique com o botão direito do mouse e selecione a opção *Add all fields*, configure a propriedade *DefaultExpression* do campo *IDEMPRESTIMO* para *0*. Selecione agora o componente *cdsEmprestimo_Item* e vincule à sua propriedade *DataSetField* ao campo *sqlEmprestimo_Item* do *cdsEmprestimo* para configurar o *Mestre-Detalhe*. Como feito no *DMPrincipal* vamos configurar o *Aggregate* para o incremento dos itens, portanto selecione o *cdsEmprestimo_Item* e altere sua propriedade *AggregatesActive* para *True*, dê um duplo clique para abrir novamente o *FieldsEditor* e com o botão direito do mouse selecione *New Field* criando um campo *Agreggate* com o nome de *Incremento*, clique sobre o novo *Field* e marque as propriedades *Visible* e *Active* para *True* e em *Expression* defina a expressão *MAX(IDEMPRESTIMO_ITEM)*.

Configure o evento AfterInsert do cdsEmprestimo setando por padrão a data do dia ao campo DATA_EMPRESTIMO no momento da inclusão, conforme é mostrado abaixo :

```
procedure TDMEmprestimo.cdsEmprestimoAfterInsert(DataSet: TDataSet);
begin
  cdsEmprestimoDATA_EMPRESTIMO.AsDateTime := Now;
end;
```



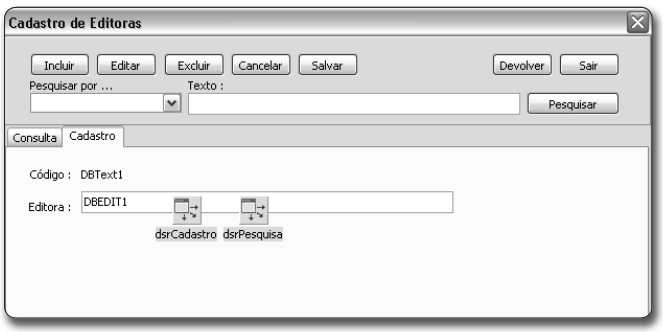


Figura 5. Exemplo do Cadastro de Editoras

Figura 6. Exemplo do Cadastro de Pessoas

Por último, iremos configurar o evento *AfterInsert* do *cdsEmprestimo_Item*, definindo o valor padrão da *DATA_PREVISTA*, bem como o auto-incremento. Observe o código abaixo :

```
procedure TDMEmprestimo.cdsEmprestimo_ItemAfterInsert(DataSet:
 TDataSet);
begin
  DMPrincipal.IncrementaValor(cdsEmprestimo_Item,'IDEMPRESTIMO_
   ITEM','Incremento');
  cdsEmprestimo_ItemDATA_PREVISTA.AsDateTime := Now;
end;
```

Com isso terminamos a implementação do último *Data Module*, vá em *File>Save All* e salve a Unit com o nome de *untDMEmprestimo*, assim iremos iniciar a última parte do nosso artigo que refere-se a implementação das entradas de dados através da nossa tela de herança.

Criação das telas da aplicação cliente

Após termos finalizado todas estas implementações chegamos a parte final do nosso artigo onde será mostrado a criação das telas e as configurações finais, como sempre, começaremos pelos cadastros mais simples. Vá em *File > New > Other > Inheritable Items* e selecione do lado direito o *frmHeranca*, deste modo será criado um novo formulário com todas as implementações realizadas na nossa herança, altre as propriedades *Name* para *frmCadEditora*, e *Caption* para *Cadastro de Editoras*. Feito isso, adicione (através do menú *File > Use Unit)* a *untDMPrincipal e aponte a propriedade DataSet do dsrPesquisa para o DMPrincipal.cdsEditoraPesquisa* e aponte para a mesma propriedade, porém do *dsrCadastro* o *DMPrincipal.cdsEditora*, selecione na *PageControl* a aba de Cadastro e adicione dois *Labels*, alterando suas propriedades *Caption* para *Código* e *Editora* respectivamente, também adicione um *DBText* e um *DBEdit* e configure suas propriedades *DataSource* para o *dsrCadastro* e em *DataField* aponte o *IDEDITORA* (para o *DBText*) e *EDITORA* (para o *DBEdit*) em seguida selecione novamente a aba de consultas. A única codificação existente no nosso formulário será o evento *OnShow* que adicionará os itens ao *ComboBox* e em seguida definirá como campo padrão de pesquisa o campo Código da Editora. Veja a implementação do evento abaixo, e o resultado final na **Figura 5**.

```
procedure TfrmCadEditora.FormShow(Sender: TObject);
begin
  inherited;
  cmbParametros.Items.Add('1 - Código');
  cmbParametros.Items.Add('2 - Editora');
  cmbParametros.ItemIndex := 0;
end;
```

Perceba como a herança tornou o nosso trabalho bem mais simplificado, exigindo pouca codificação para criarmos o nosso cadastro, clique no botão *Save All* e salve a Unit como *untCadEditora* e prossiga criando um novo *form* baseado na herança do mesmo modo como foi criado o *frmCadEditora*. Logo após ter criado o novo fomulário altere sua propriedade *Name* para *frmCadPessoa* e *Caption* para *Cadastro de Pessoas* e adicione ao evento *OnShow* as instruções de pesquisa de acordo com o código abaixo :

```
procedure TfrmCadPessoa.FormShow(Sender: TObject);
begin
  inherited;
  cmbParametros.Items.Add('1 - Código');
  cmbParametros.Items.Add('2 - Pessoa');
  cmbParametros.ItemIndex := 0;
end;
```

Adicione à cláusula *uses* do novo *form* a Unit *untDMPrincipal* para que possamos configurar os *DataSets* de pesquisa e cadastro, no componente *dsrCadastro* configure a propriedade *DataSet* vinculando-o ao *ClientDataSet cdsPessoa* localizado no *DMPrincipal*, faça o mesmo procedimento com o componente *dsrPesquisa* mas ligando-o ao *cdsPessoaPesquisa* também localizado no mesmo *Data Module*. Selecione a aba de cadastros do novo *form* e adicione sete componentes *Label*, alterando suas propriedades *Caption* respectivamente para *Código*, *Pessoa*, *Endereço*, *Bairro*, *Cidade*, *Estado* e *Telefone*, em seguida adicione um componente *DBText* configurando sua propriedade *DataSet* ao *dsrCadastro* e *DataField* ao campo correspondente, adicione seis *DBEdits* e configure-os assim como o *DBText* alterando as propriedades *DataField* à seus campos correspondentes. Após efetuar as configurações, selecione a página de consulta e salve o novo *form* nomeando-o como *untCadPessoa* . A **Figura 6** exibe o resultado final do nosso cadastro de pessoas.

Criaremos agora o nosso cadastro de livros, para isso, crie um novo *form* baseado em nossa herança e altere suas propriedades *Name* para *frmCadLivro* e *Caption* para *Cadastro de Livros* e adicione as Units *untDmPrincipal* e *untCadEditora*, prossiga configurando o evento *onShow* como mostrado a seguir:

```
  inherited;
  cmbParametros.Items.Add('1 - Código');
  cmbParametros.Items.Add('1 - Livro');
  cmbParametros.ItemIndex := 0;
```

Agora complementaremos o nosso cadastro adicionando os componentes restantes, inicie adicionando um componente DataSource e alterando suas propriedades Name para dsrItem e DataSet apontando para o cdsLivroItem, configure

Figura 7. Interface do Cadastro de Livros

a propriedade DataSet nos outros DataSources apontando dsrPesquisa para o cdsLivroPesquisa e o dsrCadastro para o cdsLivro ambos localizados no DMPrincipal. Selecione no PageControl a aba de cadastros e adicione ao form quatro Label, três DBEdit, um DBText, dois SpeedButton, uma DBGrid, e um DBNavigator. Altere as propriedades Name dos SpeedButton para spbEditora e spbGerar e Caption para "..." e Gerar, faça a configuração dos DBEdit e DBText, ligando suas propriedades DataSource ao dsrCadastro e vinculando os seus respectivos DataField. Ligue tanto o DBGrid quanto o DBNavigator ao dsrItem e adicione à DBGrid três colunas indicando os campos ITEM, STATUS e DATA_ENTRADA. Observe o resultado do visual na **Figura 7**.

Agora que fizemos as configurações necessárias iremos configurar os eventos necessários para o funcionamento correto do nosso cadastro, siga adicionando o evento *OnShow* relativo a configuração dos parâmetros de pesquisa do *ComboBox*. Como o exemplo logo o código que segue:

```
inherited;
cmbParametros.Items.Add('1 - Código');
cmbParametros.Items.Add('1 - Livro');
cmbParametros.ItemIndex := 0;
```

O código exibido na **Listagem 16** exibe o evento *OnClick* do botão *spbEditora* que referencia a chamada ao Cadastro de Editoras, a linha 04 mostra que a criação do formulário de editora só será realizada caso o *cdsLivro* esteja em estado de inserção ou edição, em seguida é criado o cadastro de editoras e o botão devolver é habilitado para a devolução do registro consultado (linha 08). Nas linhas 09 e 10 é verificado se a consulta não está vazia e o tipo de retorno do formulário chamado (que foi implementado na herança), sendo verdadeira a condição atribui os campos de pesquisa do cadastro de editora ao cadastro atual e finalmente o formulário é liberado da memória (linha 18).

Finalmente o último evento do *frmCadLivros* que refere-se a criação dos itens de livros que serão adicionados ao empréstimo posteriormente. Na **Listagem 17** temos nas linhas 06 e 07 o teste que verifica se o *cdsLivro (DataSet Mestre)* não está vazio e está em estado de inserção ou edição, e entre as linhas 09 a 14 simplesmente é feito um laço que cria os itens conforme a quantidade de livros informada. Agora que terminamos os eventos selecione a aba de consultas e salve a nova Unit como *untCadLivro* para prosseguirmos com o último cadastro do nosso projeto.

Novamente crie um novo *form* baseado na herança, este por sinal é o último e mais complexo dos cadastros de nossa aplicação. Inicie alterando as suas propriedades *Name* para *frmCadEmprestimo* e *Caption* para *Empréstimos*, em seguida adicione as Units *untCadPessoa*, *untCadLivro* e *untDMEmprestimo*, clique no botão *spbExcluir* e altere sua

Listagem 16. Evento OnClick do SpeedButton spbEditora

```
01  procedure TfrmCadLivro.spbEditoraClick(Sender: TObject);
02  begin
03    inherited;
04    if DMPrincipal.cdsLivro.State in [dsEdit, dsInsert] then
05      begin
06        frmCadEditora := TfrmCadEditora.Create(Self);
07        try
08          frmCadEditora.spbDevolver.Visible := True;
09          if (frmCadEditora.ShowModal = mrOk) and
10             (not frmCadEditora.dsrPesquisa.DataSet.IsEmpty) then
11            begin
12              DMPrincipal.cdsLivroIDEDITORA.AsInteger :=
13              frmCadEditora.dsrPesquisa.DataSet.
                FieldByName('IDEDITORA').AsInteger;
14              DMPrincipal.cdsLivroEDITORA.AsString :=
15              frmCadEditora.dsrPesquisa.DataSet.
                FieldByName('EDITORA').AsString;
16            end;
17        finally
18          FreeAndNil(frmCadEditora);
19        end;
20      end;
21  end;
```

Listagem 17. Evento OnClick do SpeedButton spbGerar

```
01  procedure TfrmCadLivro.spbGerarClick(Sender: TObject);
02  var
03    i : Integer;
04  begin
05    inherited;
06    if (not DMPrincipal.cdsLivro.IsEmpty) and
07       (DMPrincipal.cdsLivro.State in [dsEdit, dsInsert]) then
08      begin
09        for i := 0 to DMPrincipal.cdsLivroQUANTIDADE.AsInteger - 1 do
10          begin
11            if not (DMPrincipal.cdsLivroItem.State in
                [dsInsert]) then
12              DMPrincipal.cdsLivroItem.Append;
13              DMPrincipal.cdsLivroItem.Post;
14          end;
15      end;
16  end;
```

propriedade *Visbile* para *False*, pois não será possível realizar exclusões de empréstimos. Agora adicionaremos os componentes restantes para a implementação do mesmo, comece adicionando dois *DataSource* e alterando suas propriedades *Name* para *dsrItem* e *dsrLivroPesquisa* em seguida ligue respectivamente suas propriedades *DataSet* a *cdsItem* e *cdsBuscaLivros*, configure o *dsrPesquisa* apontando-o para o *cdsEmprestimoPesquisa* e o *dsrCadastro* apontando para o *cdsEmprestimo* todos localizados no *Data Module DMEmprestimo*. Selecione a aba de cadastros e adicione cinco *Label*, três *DBEdit*, um *Edit*, um *DBText*, três *SpeedButton*, dois *DBGrid* e um *DBNavigator*, altere as propriedades *Caption* dos *Label* para *Código*, *Pessoa*, *Data do Empréstimo*, *Código do Livro* e *Itens*, altere as propriedades *Name* do *Edit* para *edtPesquisaLivro* e limpe a propriedade *Text*, nos *SpeedButton* altere *Name* para *spbPessoa*, *spbPesquisarLivro* e *spbAdicionar* e *Caption* para *"..."*, *Pesquisar Livro* e *Adicionar - >*. Ligue as propriedades *DataSource* dos *DBEdits* e do *DBText* no *dsrCadastro* e configure seus respectivos *DataField*. Aponte no primeiro *DBGrid* o *DataSource dsrLivroPesquisa* e adicione as colunas *ITEM* e *STATUS*, último configure o outro *DBGrid* e o *DBNavigator* apontando suas propriedades *DataSource* ao *dsrItem*, na DBGrid adicione as colunas relativas ao *IDLIVRO_ITEM*, *LIVRO*, *DATA_PREVISTA* e *DATA_DEVOLUCAO*, por último selecione o DBNavigator e altere sua propriedade *VisibleButtons*, marcando o *nbInsert* como *False*. Agora vamos codificar um pouco o nosso formulário.

Começaremos definindo os campos de pesquisa do empréstimo no evento *OnShow* como o exemplo a seguir:

```
inherited;
cmbParametros.Items.Add('1 - Empréstimo');
cmbParametros.Items.Add('2 - Código da Pessoa');
cmbParametros.Items.Add('3 - Nome da Pessoa');
cmbParametros.ItemIndex := 0;
```

Após a configuração dos parâmetros de pesquisa iremos implementar o evento *OnClick* do *SpeedButton spbPessoa* como mostra a **Listagem 18.** Se observarmos, o código é muito semelhante ao que foi implementado no cadastro de editoras, sendo a única diferença a chamada correspondente ao cadastro de pessoas.

Dê um clique no *Edit edtPesquisaLivro* e codifique o evento *OnKeyPress* de acordo com a **Listagem 19.** Este código é responsável por selecionar todos os Itens de livros disponíveis para empréstimo, repare que primeiramente é verificado se a tecla pressionada está entre números de 0 a 9, *BackSpace* ou se foi pressionado o *Enter*, caso contrário a tecla é anulada (linhas 05 e 06). Caso a tecla pressionada seja *Enter* a busca é feita passando como parâmetro o texto digitado no *edtPesquisaLivro* (linhas 09 a 15).

A outra forma de buscar os livros é exibida na **Listagem 20**, onde temos a chamada ao Cadastro de Livros. Repare que o evento é o mesmo dos codificados anteriormente, porém é declarada uma variável do tipo Char (linha 02) , esta mesma é inicializada na linha 07 com o *char #13 (*tecla *Enter)* para que ao retornar o registro corresponde da seleção do cadastro de livros, seja atribuido o valor do campo *IDLIVRO* ao *edtPesquisaLivro* (linhas 13 e 14), e na linha 15 temos a chamada ao evento *OnKeyPress* do *edtPesquisaLivro* passando como parâmetro a tecla *Enter* e fazendo com que seja simulado o pressionamento da mesma, efetuando assim a consulta de livros.

Por último implementaremos a inserção dos itens de livros para os itens de empréstimo por meio do evento *OnClick* do *SpeedButton* que pode ser observado na **Listagem 21.** Inicialmente criamos uma variável do tipo inteiro (linha 02) na qual será responsável por armazenar a quantidade de registros repetidos, dessa forma antes de inserirmos vamos verificar se o item em questão já existe no empréstimo, garantindo assim

Listagem 18. Evento OnClick do SpeedButton spbPessoa

```
01  procedure TfrmCadEmprestimo.spbPessoaClick(Sender: TObject);
02  begin
03    if DMEmprestimo.cdsEmprestimo.State in [dsEdit, dsInsert] then
04      begin
05        frmCadPessoa := TfrmCadPessoa.Create(Self);
06        try
07          frmCadPessoa.spbDevolver.Visible := True;
08          if (frmCadPessoa.ShowModal = mrOk) then
09            begin
10              DMEmprestimo.cdsEmprestimo.
                 FieldByName('IDPESSOA').AsInteger :=
11               frmCadPessoa.dsrPesquisa.DataSet.
                 FieldByName('IDPESSOA').AsInteger;
12              DMEmprestimo.cdsEmprestimo.
                 FieldByName('PESSOA').AsString :=
13               frmCadPessoa.dsrPesquisa.DataSet.
                 FieldByName('PESSOA').AsString;
14            end;
15        finally
16          FreeAndNil(frmCadPessoa);
17        end;
18      end;
19  end;
```

Listagem 19. Evento OnKeyPress do Edit edtPesquisaLivro

```
01  procedure TfrmCadEmprestimo.edtPesquisaLivroKeyPress(Sender:
     TObject;
02    var Key: Char);
03  begin
04    inherited;
05    if not (Key  in ['0'..'9', #13, #8]) then
06      Key := #0
07    else
08      begin
09        if Key = #13 then
10          begin
11            DMEmprestimo.cdsBuscaLivros.Close;
12            DMEmprestimo.cdsBuscaLivros.DataRequest(
13            VarArrayOf([edtPesquisaLivro.Text]));
14            DMEmprestimo.cdsBuscaLivros.Open;
15          end;
16      end;
17  end;
```

Listagem 20. Evento OnClick do SpeedButton spbPesquisarLivro

```
01  procedure TfrmCadEmprestimo.spbPesquisarLivroClick(Sender:
     TObject);
02  var Key : Char;
03  begin
04    inherited;
05    if DMEmprestimo.cdsEmprestimo.State in [dsEdit, dsInsert] then
06      begin
07        Key := #13;
08        frmCadLivro := TfrmCadLivro.Create(Self);
09        try
10          frmCadLivro.spbDevolver.Visible := True;
11          if (frmCadLivro.ShowModal = mrOk) then
12            begin
13              frmCadEmprestimo.edtPesquisaLivro.Text :=
14                frmCadLivro.dsrPesquisa.DataSet.
                   FieldByName('IDLIVRO').AsString;
15              frmCadEmprestimo.edtPesquisaLivroKeyPress(Sender, Key);
16            end;
17        finally
18          FreeAndNil(frmCadLivro);
19        end;
20      end;
21  end;
```

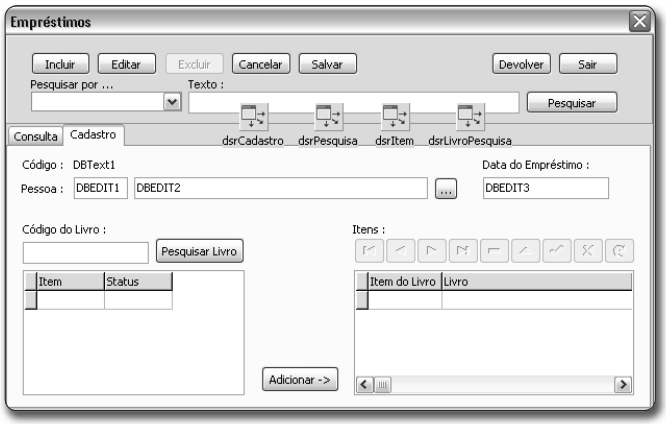


Figura 8. Interface da tela de Empréstimos

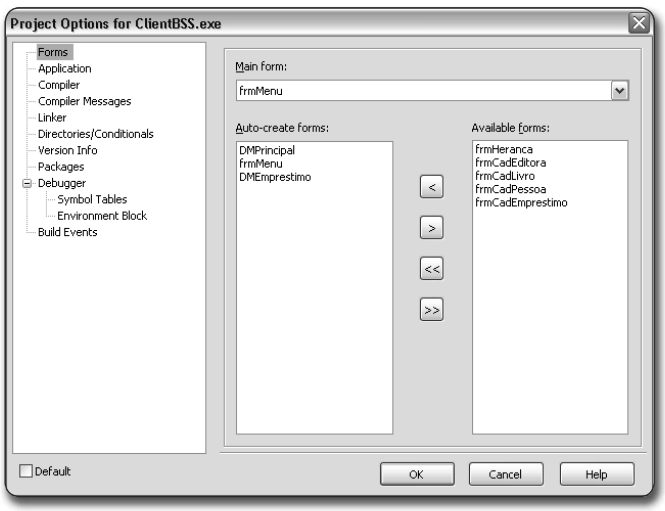


Figura 9. Configuração dos formulários e Data Modules de inicialização

que as validações sejam feitas na camada cliente e enviando os dados já prontos para serem gravados. Antes de verificar se o item já foi adicionado é necessário verificar se o empréstimo está ativo pois caso contrário será gerado um erro. Após a verificação inicial, o filtro é desabilitado (linha 11) e preparada de acordo com o registro que deve ser adicionada (linhas 12 e 13), no momento em que o filtro é aplicado, a variável *RecCount* recebe o valor de registros contidos no filtro. Com a quantidade de registros alocada na variável, o filtro é desativado (linha 16) e em seguida verifica-se se houveram registros pertencentes a esta validação, sendo afirmativo a mensagem de erro é exibida (linhas 19 e 20), caso contrário a inserção é realizada normalmente (linhas 25 a 31).

Feito isso, selecione a aba de consultas na *PageControl* e salve a Unit com o nome de *untCadEmprestimo*. O resultado final do nosso formulário pode ser visto na **Figura 8.**

Para finalizar é necessário configurarmos quais serão os formulários e *Data Modules* que inicializarão na nossa

Listagem 21. Evento OnClick do SpeedButton spbAdicionar

```
01  procedure TfrmCadEmprestimo.spbAdicionarClick(Sender: TObject);
02  var RecCount : Integer;
03  begin
04    inherited;
05    with DMEmprestimo do
06      begin
07        RecCount := 0;
08        if (cdsEmprestimo_Item.Active) then
09          begin
10            if cdsEmprestimo_Item.Filtered then
11              cdsEmprestimo_Item.Filtered := False;
12              cdsEmprestimo_Item.Filter := 'IDLIVRO_ITEM = ' +
13              cdsBuscaLivros.FieldByName('IDLIVRO_ITEM').AsString;
14              cdsEmprestimo_Item.Filtered := True;
15              RecCount := cdsEmprestimo_Item.RecordCount;
16              cdsEmprestimo_Item.Filtered := False;
17            if RecCount > 0  then
18              begin
19                Application.MessageBox('Este livro já foi adicionado !',
20                  'Aviso !', MB_OK + MB_ICONWARNING);
21                edtPesquisaLivro.SetFocus;
22              end
23            else
24              begin
25                if not (cdsEmprestimo_Item.State in [dsInsert]) then
26                  cdsEmprestimo_Item.Insert;
27                  cdsEmprestimo_ItemIDLIVRO_ITEM.AsInteger :=
28                  cdsBuscaLivros.FieldByName('IDLIVRO_ITEM').
                      AsInteger;
29                  cdsEmprestimo_ItemLIVRO.AsString :=
30                  cdsBuscaLivros.FieldByName('LIVRO').
                      AsString;
31                  cdsEmprestimo_Item.Post;
32              end;
33          end;
34      end;
35  end;
```

aplicação. Siga o menú *Project* > *Options* e clique esquerdo na guia Forms, configure sua aplicação como mostra a **Figura 9**.

Conclusão

Como vimos nessa série de artigos a arquitetura de multicamadas pode nos proporcionar inúmeras vantagens em relação à arquitetura cliente-servidor, devido a sua alta performance em ambientes distribuídos, centralização de regras, alta disponibilidade e facilidade de desenvolvimento com os mais diferenciados bancos de dados. Contudo, é necessário que se estude as diferentes maneiras de se utilizar esta tecnologia, adequando-as conforme a necessidade de seu ambiente e aplicação. Espero que todos tenham gostado. Um forte abraço e até a próxima. ●

Dê seu feedback sobre esta edição!

A Clubedelphi tem que ser feita ao seu gosto. Para isso, precisamos saber o que você, leitor, acha da revista!

Dê seu voto sobre este artigo, através do link:

www.devmedia.com.br/clubedelphi/feedback

Nesta seção você encontra artigos intermediários sobre Delphi Win 32 e Delphi for .NET

tiOPF na Prática

Faça um controle de cheques usando orientação a objetos – Parte II

Nesse artigo veremos
- Uma introdução ao framework tiOPF;
- Como utilizá-lo em um sistema comum;
- Como gerar relatórios.

Qual a finalidade
Mostrar o uso adequado de ferramentas que auxiliam no desenvolvimento orientado a objetos e como aplicá-las no dia a dia.

Quais situações utilizam esse recurso?
No desenvolvimento de sistemas em geral.

Resumo do DevMan

O tiOPF é um dos mais antigos e estáveis frameworks ORM, Object/Relational Mapper, para Delphi. É utilizado por grandes empresas no exterior e, neste artigo vamos continuar sua utilização no desenvolvimento do sistema de controle de cheques. Será visto como gerar relatórios baseando-se nos objetos existentes.

Paulo Quicoli
pauloquicoli@devmedia.com.br
Editor técnico da revista ClubeDelphi, formado em processamento de dados pela FATEC-TQ. Atua como analista e desenvolvedor da ControlM Informática (www.controlm.com.br). Utiliza Delphi desde sua primeira versão, entusiasta do desenvolvimento orientado a objetos tem publicado vários artigos sobre o assunto. Blog: www.pauloquicoli.spaces.live.com .

No artigo anterior, vimos que o *tiOPF* é um *framework* que realiza o mapeamento objeto relacional para nós, assumindo a responsabilidade de persistir objetos de negócio em tabelas de um banco de dados. Quando desenvolvemos um sistema aplicando a orientação a objetos nos deparamos com um problema chamado *impedância objeto/relacional*. Resumidamente esse problema representa a diferença existente entre esses dois mundos.

Par exemplificar, iniciamos um projeto com o objetivo de controlar cheques usando orientação a objetos. Continuando com o projeto, nos resta manter o registro dos cheques, permitir que uma consulta seja feita e geração de um relatório. Vamos agora criar nossa classe que representa os cheques. Como já citado na primeira parte deste artigo, o *tiOPF* oferece para nós uma estrutura básica pronta para inserção, exclusão e consulta de objetos que é realizada

através de coleções de objetos, representadas por classes do tipo *TtiObjectList*.

Retornando ao projeto

Caso não tenha tido acesso ao artigo da edição anterior ou mesmo não tenha desenvolvido o exemplo citado na edição, o ideal é que você faça o download do código-fonte no site da DevMedia e continue de onde paramos para que possamos entender de forma mais tranqüila o que veremos nessa edição. O código-fonte pode se obtido no link *www.devmedia.com.br/resumo/download.asp?site=3&idrevista=126*.

Siga abrindo o arquivo produzido no artigo anterior e logo em seguida a *unit BOM.pas*. Nós vamos criar uma nova classe chamada *TCheques*, como ilustra a **Listagem 1**.

Listagem 1. Classe TCheque

```
01  TCheques = class;
02  TCheque = class;
03
04  TCheques = class(TtiObjectList)
05  private
05 protected
06    function GetItems(i: integer): TCheque; reintroduce;
07    procedure SetItems(i: integer; const Value: TCheque); reintroduce;
08 public
09    property Items[i:integer]: TCheque read GetItems write SetItems;
10    procedure Add(AObject: TCheque); reintroduce;
11    procedure Clear; override;
12    procedure Save; override;
13  end;
```

Nas linhas 1 e 2 declaramos antecipadamente as duas classes que usaremos. Veja na linha 4 que criamos a classe *TCheques* herdando de *TtiObjectList*, que é base para lista de objetos oferecendo métodos para salvar e carregar as informações. Sua implementação também é bem simples, como mostra a **Listagem 2**.

Listagem 2. Implementação da classe Tcheques

```
01 procedure TCheques.Add(AObject: TCheque);
02 begin
03   inherited Add(AObject);
04 end;
05
06 procedure TCheques.Clear;
07 begin
08   inherited;
09   ObjectState:= posEmpty;
10 end;
11
12 function TCheques.GetItems(i: integer): TCheque;
13 begin
14   result:= TCheque(inherited GetItems(i));
15 end;
16
17 procedure TCheques.Save;
18 begin
19   inherited;
20
21 end;
22
23 procedure TCheques.SetItems(i: integer; const Value: TCheque);
24 begin
25   inherited SetItems(i, Value);
26 end;
```

Semelhantemente à classe *TContas*, criada no artigo anterior, esta também é uma coleção de objetos e nada mais lógico do que ela oferecer meios para se adicionar objetos. É exatamente isso que o método *Add* faz. O método *Clear* por sua vez marca o estado a coleção em *posEmpty*, que indica ao *framework* que a mesma deve ser esvaziada. Vamos agora definir a classe *TCheque*, como mostra a **Listagem 3**.

Listagem 3. Classe TCheque

```
01  TCheque = class(TtiObject)
02  private
03    FDescricao: string;
04    FEmissao: TDateTime;
05    FVencimento: TDateTime;
06    FNumero: string;
07    FValor: Currency;
08    FConta: TConta;
09    procedure SetDescricao(const Value: string);
10    procedure SetEmissao(const Value: TDateTime);
11    procedure SetVencimento(const Value: TDateTime);
12    procedure SetNumero(const Value: string);
13    procedure SetValor(const Value: Currency);
14    procedure SetConta(const Value: TConta);
15    function GetContaOIDAsString: String;
16    procedure SetContaOIDAsString(const Value: String);
17    function GetContaGuiAsString: string;
18  protected
19    function GetOwner: TCheques; reintroduce;
20    procedure SetOwner(const Value: TCheques); reintroduce;
21  public
22    constructor Create; override;
23    property Owner: TCheques read GetOwner write SetOwner;
24    property Conta: TConta read FConta write SetConta;
25  published
26    property Descricao: string read FDescricao write SetDescricao;
27    property Emissao: TDateTime read FEmissao write SetEmissao;
28    property Vencimento: TDateTime read FVencimento write
       SetVencimento;
29    property Numero: string read FNumero write SetNumero;
30    property Valor: Currency read FValor write SetValor;
31    property ContaOIDAsString: String read GetContaOIDAsString
       write SetContaOIDAsString;
32    property ContaGuiAsString: string read GetContaGuiAsString;
33  end;
```

Observe que novamente as propriedades estão na seção *published*, contudo temos duas na seção *public*. A propriedade *Owner* é utilizada para indicar em qual lista de objetos ele está inserido. A propriedade *Conta*, por sua vez representa a conta que está associada ao cheque.

Temos também a propriedade *ContaGuiAsGui* que é somente leitura. Ela foi criada para facilitar a exibição dos dados em um formulário. Seu valor de retorno é a soma das propriedades *Banco, Agencia e Conta*, como pode ser visto no código a seguir.

```
  if FConta = nil then
     result := ''
  else
     result := FConta.Banco + ' ' + FConta.Agencia+ '-'+ FConta.
Conta;
```

Outra propriedade que precisa ser mencionada é a *ContaOIDAsString*, é nela que iremos armazenar o valor da chave primária de uma conta que está relacionada a um cheque. Ao *setarmos* esse valor realizamos a carga do objeto *TConta* correspondente e o passamos para a propriedade *Conta*. Isso é visto no código a seguir.

```
procedure TCheque.SetContaOIDAsString(const Value: String);
begin
  FConta:= SingletonContas.Find(Value);
end;
```

Observe que estamos fazendo uso de um elemento ainda não citado aqui, o objeto *SingletonContas*. Esse objeto representa o padrão de projeto *singleton*, representando um ponto

único e global para recuperar objetos do tipo *TConta*. Sua implementação pode ser vista na **Listagem 4.**

Listagem 4. SingletonContas

```
01 function SingletonContas: TContas;
02
03 implementation
04
05 var
06  _ListaContas: TContas;
07
08 function SingletonContas: TContas;
09 begin
10   if _ListaContas = nil then
11   begin
12     _ListaContas := TContas.Create;
13     _ListaContas.Read
14   end;
15   result := _ListaContas;
16 end;
```

Veja que a função simplesmente retorna uma instância de *TContas*, que está armazenada na variável global *_ListaContas*, verificando se já não existe uma instância. Com isso pronto podemos agora criar a tabela no banco de dados que irá armazenar os cheques, como mostra a **Listagem 5.**

Listagem 5. Tabela CHEQUE

```
CREATE TABLE CHEQUE (
    DESCRICAO   VARCHAR(500),
    EMISSAO     DATE,
    VENCIMENTO  DATE,
    NUMERO      VARCHAR(6),
    VALOR       NUMERIC(15,2),
    ID_CHEQUE   VARCHAR(36) NOT NULL,
    ID_CONTA    VARCHAR(36) NOT NULL
);

ALTER TABLE CHEQUE ADD CONSTRAINT PK_CHEQUE PRIMARY KEY (ID_CHEQUE);
ALTER TABLE CHEQUE ADD CONSTRAINT FK_CHEQUE_CONTA FOREIGN KEY
 (ID_CONTA) REFERENCES CONTA (ID_CONTA);
```

Agora que temos a classe pronta e seu banco de dados pronto, é necessário informar isso ao *tiOPF*. Vamos fazer o mapeamento objeto relacional, adicionando o código da **Listagem 6** ao procedimento *RegisterAutoMappings*.

Listagem 6. Mapeamento de TCheque

```
01    RegisterMapping(TCheque,'CHEQUE','OID', 'ID_CHEQUE', [pktDB] );
02    RegisterMapping(TCheque,'CHEQUE','Descricao','DESCRICAO');
03    RegisterMapping(TCheque,'CHEQUE','Emissao','EMISSAO');
04    RegisterMapping(TCheque,'CHEQUE','Vencimento','VENCIMENTO');
05    RegisterMapping(TCheque,'CHEQUE','Numero','NUMERO');
06    RegisterMapping(TCheque,'CHEQUE','Valor','VALOR');
07    RegisterMapping(TCheque,'CHEQUE','ContaOIDAsString','ID_CONTA');
08    RegisterCollection(TCheques, TCheque);
```

Veja que a propriedade *ContaGuiAsString* não foi mapeada, visto que ela é somente leitura.

Criando o cadastro de cheques

Assim como foi feito no artigo anterior, vamos adicionar um novo formulário que herde de *TFormTIPerEditDialog*. Vamos chamá-lo de *TEditChequesF* e incluir os seguintes componentes visuais do *tiOPF:* um *TtiPerAwareComboBoxDynamic, dois TtiPerAwareDateTimePicker, dois TtiPerAwareEdit e um TtiPerAwareMemo*. Organize e configure os controles para que fiquem como na **Figura 1**.

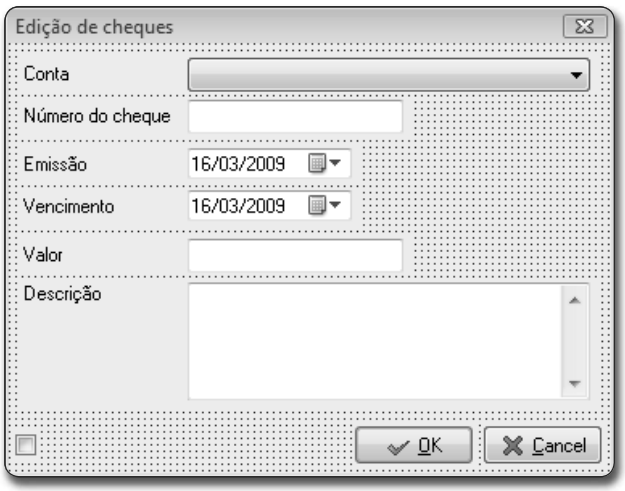


Figura 1. Formulário de edição de cheques

Agora vamos sobrescrever o método *SetData* como mostra a **Listagem 7**.

Listagem 7. Ligando os dados do objeto

```
01 public
02  { Public declarations }
03  procedure SetData(const Value: TtiObject); override ;
04
05 procedure TEditChequesF.SetData(const Value: TtiObject);
06 begin
07   inherited;
08   comboContas.Value  := TCheque(DataBuffer).Conta;
09   editNumero.LinkToData(DataBuffer,'Numero');
10   editEmissao.LinkToData(DataBuffer,'Emissao');
11   editVencimento.LinkToData(DataBuffer,'Vencimento');
12   editValor.LinkToData(DataBuffer,'Valor');
13   editDescricao.LinkToData(DataBuffer,'Descricao');
14 end;
```

Como pode ser visto, os controles de edição do *tiOPF* oferecem recursos para ligarmos objetos. Veja que estamos até mesmo setando o valor para o controle *comboContas*, que por sua vez deve exibir uma lista de contas disponíveis.

Exibindo as contas disponíveis

A idéia para a tela de cadastro de cheques é que ao entrar nela, possamos escolher no *ComboBox* de qual conta o cheque pertence, formando assim uma espécie de *lookup*. A primeira coisa a se fazer é selecionar esse *ComboBox (TtiPerAwareComboBoxDynamic)* e indicar para ele qual propriedade será exibida em sua lista. Isso é feito em sua propriedade *FieldNameDisplay*, na qual informamos o atributo *ContaAsGui*. Só isso não basta, é preciso preencher esse *ComboBox* com objetos. Isso é feito no evento *OnCreate* do formulário, como pode ser visto na **Listagem 8**.

ClubeDelphi PLUS www.devmedia.com.br/clubedelphi/portal.asp

Acesse agora mesmo o portal do assinante ClubeDelphi e veja uma vídeo aula de Luciano Pimenta que mostra como utilizar o IBExpert para administração avançada de banco de dados Firebird/Interbase.

http://www.devmedia.com.br/articles/viewcomp.asp?comp=3082

Listagem 8. Carregando as contas disponíveis

```
01 procedure TEditChequesF.FormCreate(Sender: TObject);
02 begin
03   inherited;
04   SingletonContas.Read;
05   comboContas.List:= SingletonContas.List;
06 end
```

Mais uma vez fazemos uso do *SingletonContas* para recuperar todas as contas disponíveis. Isso é feito ao executarmos o método *Read*. Posteriormente passamos para a lista de objetos do *comboContas* a lista dos objetos retornados. Agora ao criar um novo cheque, como podemos passar para esse cheque qual conta foi selecionada? Através do evento *onChange* do *comboContas* podemos codificar isso, como pode ser visto no código a seguir.

```
TCheque(DataBuffer).Conta := ComboContas.Value as TConta;
```

O componente *TtiPerAwareComboBoxDynamic* possui a propriedade *Value*, que retorna um objeto do tipoTObject representando o item que está selecionado, por isso é necessário realizar um *cast* para a classe *TConta*.

Exibindo o formulário de cadastro

Muito bom, até agora fizemos toda a programação das classes e das funções que precisamos no sistema, mas é preciso exibir esse formulário. Para isso adicionamos ao menu principal, do formulário principal, o item de menu *Cadatros>Cheques*. Em seu evento *onClick* codificamos como mostra a **Listagem 9**.

Listagem 9. Exibindo a tela de cadastro de cheques

```
01 procedure TPrincF.Cheques1Click(Sender: TObject);
02 var
03   lCheque: TCheque;
04   lCheques: TCheques;
05 begin
06   lCheque:= TCheque.CreateNew;
07   if TEditChequesF.Execute(lCheque) then
08   begin
09    lCheques := TCheques.Create;
10    lCheques.Add(lCheque);
11    lCheques.Save;
12    FreeAndNil(lCheque);
13   end
14   else
15    lCheque.Free;
16 end;
```

Criamos um novo objeto *TCheque* e já o passamos como parâmetro ao método *Execute*,que por sua vez irá exibir o formulário, retornando ainda a ação tomada pelo usuário, indicando se o cadastro foi confirmado ou cancelado. Execute a aplicação e insira alguns cheques.

Consultando os cheques

Até o momento podemos cadastrar cheques. Um dos requisitos deste pequeno sistema é poder efetuar uma consulta dos cheques que nós emitimos. Essa consulta pode ser baseada na data de emissão ou vencimento do cheque, de tal forma que poderemos saber o montante em reais (R$) do que foi emitido. Também vamos imprimir o resultado dessa consulta. Na **Figura 2** vemos como deve ficar nosso formulário de pesquisa.

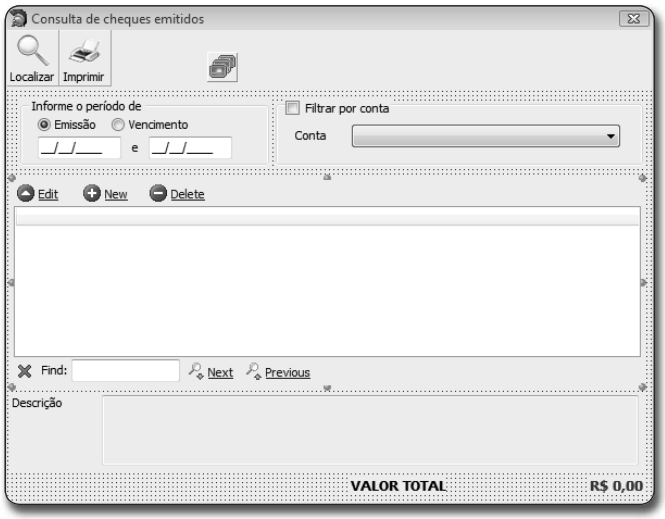


Figura 2. Formulário de pesquisa

Adicione ao projeto um novo formulário, salvando-o como *ConsultaChequesF*. Os controles diferentes que vamos adicionar são: um *TtiPerAwareComboBoxDynamic*, que permitirá um filtro por conta, um *TtiVTListView* que exibirá o resultado da consulta e um *TtiPerAwareMemo* que mostrará a descrição do cheque selecionado no *ListView*. O restante dos controles já é conhecido e não precisam ser citados.

O *tiOPF* oferece um meio de realizar as pesquisas de uma forma muito prática através de critérios de consulta. Esses critérios são propriedades que representam operadores lógicos como *AND, OR, BETWEEN* e que posteriormente são transformados em sentenças *SQL*. Esses critérios estão implementados na *unit tiCriteria* e são utilizados em um tipo especial de lista de objetos, o tipo *TtiFilteredObjectList*. Vamos criar uma classe que herde desse tipo para que possamos realizar consultas de cheques.

Na *unit BOM.pas* vamos adicionar a classe *TFiltroCheque*, como mostra a **Listagem 10**.

Listagem 10. Classe TFiltroCheque e sua implementação

```
01  TFiltroCheques = class(TtiFilteredObjectList)
02   public
03     procedure Read; override;
04     procedure Save; override;
05   end;
06
07  procedure TFiltroCheques.Read;
08  begin
09    inherited;
10
11  end;
12
13  procedure TFiltroCheques.Save;
14  begin
15    inherited;
16
17  end;
```

É necessário agora informar ao *tiOPF* que essa classe será uma lista de cheques. Isso é feito no mapeamento, método *RegisterAutoMappings*, registrando essa também essa lista para a classe *TCheque*, como mostra o código a seguir.

```
RegisterCollection(TFiltroCheques, TCheque);
```

Agora podemos voltar ao formulário de consulta e nele referenciar em uma variável essa classe de consulta. Adicione na seção *private* um campo chamado *FListaCheques* do tipo *TFiltroCheques*, lembrando que é preciso adicionar referência às *units BOM e tiCriteria*. Nós instanciamos essa lista de objetos no evento *OnCreate* do formulário, veja a seguir:

```
procedure TConsultaChequesF.FormCreate(Sender: TObject);
begin
  FListaCheques := TFiltroCheques.Create;
end;
```

Depois, no evento *OnShow* nós definimos a ligação dos controles com o objeto, como pode ser visto na **Listagem 11**.

Listagem 11. Ligando os controles

```
01 procedure TConsultaChequesF.FormShow(Sender: TObject);
02 begin
03   editData1.Text := DateToStr(Now);
04   editData2.Text := DateToStr(Now);
05   comboContas.List:= SingletonContas.List;
06
07   ListView.AddColumn('ContaGuiAsString', vttkString, 'Banco', 150);
08   ListView.AddColumn('Numero', vttkString, 'Número', 80);
09   ListView.AddColumn('Emissao', vttkDate, 'Emissão', 80);
10   ListView.AddColumn('Vencimento', vttkDate, 'Vencimento', 80);
11   ListView.AddColumn('Valor', vttkCurrency, 'Valor', 80);
12
13   ListView.Data:= nil;
14   LabelTotal.Caption := 'R$ 0,00';
15 end;
```

Observe que estamos obtendo o conteúdo do *comboContas* a partir do *Singleton* previamente criado. Veja que estamos definindo também as colunas que estarão visíveis no *ListView*. Ao passarmos *NIL* para a propriedade *Data* do *ListView*, seu conteúdo é eliminado.

Vamos codificar a consulta. Dê um clique duplo no botão *Localizar* e digite o código da **Listagem 12**.

Listagem 12. Realizando a consulta

```
01 procedure TConsultaChequesF.btLocalizarClick(Sender: TObject);
02 var
03  propriedade: string;
04 begin
05   propriedade := 'Emissao';
06   if rbVencimento.Checked then
07     propriedade := 'Vencimento';
08
09   FListaCheques.Clear;
10   FListaCheques.Criteria.ClearAll;
11   FListaCheques.Criteria.AddBetween(propriedade,StrToDate
      (EditData1.Text),StrToDate(EditData2.Text));
12   if (ckFiltrarPorConta.Checked) and (comboContas.ItemIndex> -1)
      then
13      FListaCheques.Criteria.AddEqualTo('ContaOIDAsString',
         TConta(ComboContas.Value).OID.AsString);
14
15   FListaCheques.Read;
16   ListView.Data := FListaCheques;
17   ListView.Refresh;
18   SomarTotais;
19 end;
```

Primeiramente nós identificamos qual *RadioButton* está selecionado, para sabermos se vamos consultar por emissão ou vencimento dos cheques. Veja que o campo *FListaCheques* possui uma propriedade chamada *Criteria*, que é uma lista de critérios de busca. Antes de adicionar qualquer critério chamamos o método *ClearAll* que limpa da lista qualquer critério utilizando anteriormente, garantindo assim que uma nova consulta seja realizada.

Na seqüência adicionamos o critério *between* através do método *AddBetween*, onde passados as datas que o usuário digita nos campos *EditData1* e *EditaData2*. Depois verificamos se é para realizar um filtro por conta também. Isso porque nosso sistema permite gerenciar várias contas.

Verificamos se o *CheckBox ckFiltrarPorConta* está marcado e se foi selecionada alguma conta no *ComboBox comboContas*, caso tudo isso seja verdadeiro adicionamos outro critério de busca, o *EqualTo*, através do método *AddEqualTo* que recebe como parâmetro o nome da propriedade e o valor que a mesma deve possuir. *EqualTo* significa *igual à*, ou seja ao adicioná-lo estamos procurando por cheques que tenham a propriedade *ContaOIDAsString* igual ao *OID* do objeto selecionado no *ComboContas*. Em seguida executamos a consulta chamando o método *Read*. Internamente esses critérios são transformados em sentenças *SQL* e são enviados ao banco de dados.

Para exibir o resultado dessa consulta passamos para a propriedade *Data* do *ListView* o próprio objeto *FListaCheques* e pedimos que ele se atualize, pelo método *Refresh*.

Além de exibir os cheques incluí uma funcionalidade que mostra a soma dos valores dos cheques resultantes da consulta. Assim podemos saber, além da quantidade de cheque emitidos, o quanto foi gasto. Isso está implementado no método *SomarTotais* que pode ser visto a seguir.

```
procedure TConsultaChequesF.SomarTotais;
begin
  FSoma := 0;
  FListaCheques.ForEach(SomaTotais);
  LabelTotal.Caption := FloatToStrF(FSoma, ffCurrency, 15, 2);
end;
```

Temos nele um campo privado FSoma que armazenará o cálculo que é realizado pelo método *ForEach* de *FListaCheques*. Esse método é muito interessante pois ele recebe como parâmetro um método. Isso sim, um método que será executado a cada elemento que *FListaCheques* possui. Veja que passamos como parâmetro o método *SomaTotais* que veremos a seguir.

```
procedure TConsultaChequesF.SomaTotais(AObject: TtiObject);
begin
  FSoma := FSoma + TCheque(AObject).Valor;
end;
```

Observe que o método *SomaTotais* recebe como parâmetro um objeto do tipo *TtiObject* e que estamos utilizando um *cast* para objeto o valor do cheque e efetuar a soma. Após a soma ter sido feita, exibimos seu resultado na *Label LabelTotal*.

Se executarmos a consulta nesse momento poderemos ver que a mesma está funcionando, contudo a descrição do cheque não é exibida. Para isso ser feito vamos codificar o evento *OnItemArrive* do *ListView* como a seguir.

```
procedure TConsultaChequesF.ListViewItemArrive(pVT:
 TtiCustomVirtualTree;AData: TtiObject; AItem: PVirtualNode);
begin
  editDescricao.LinkToData(AData,'Descricao');
end;
```

Este evento é disparado sempre que um item é selecionado na lista. Esse item é representado pelo parâmetro *AData*, portanto apenas ligamos o controle *editDescricao* à propriedade *Descricao* do item que foi selecionado. Execute novamente a aplicação e veja que agora vemos as descrições de cada cheque.

É interessante também que permitamos que novos cheques possam ser incluídos a partir desse formulário mesmo, bem como uma edição ou exclusão de algum cheque existente. Fazemos a inclusão através do evento *OnItemInsert* do *ListView*, como mostra a **Listagem 13**.

Listagem 13. Incluindo um novo cheque

```
01 procedure TConsultaChequesF.ListViewItemInsert
    (pVT: TtiCustomVirtualTree;
02   AData: TtiObject; AItem: PVirtualNode);
03 var
04   obj : TCheque ;
05 begin
06   obj:= TCheque.CreateNew;
07   if EditChequesF.Execute(obj) then
08   begin
09     FListaCheques.Add(obj);
10     FListaCheques.Save;
11     ListView.Refresh(obj);
12     SomarTotais;
13   end
14   else
15     obj.Free ;
16 end;
```

Semelhantemente como fizemos no formulário principal, instanciamos um novo objeto *TCheque* e o passamos para o formulário *EditChequesF*. A diferença aqui é que após ter sido incluído realizamos um *refresh* no *ListView* para que o mesmo exiba o novo cheque cadastrado e chamamos o método *SomarTotais* para recalcular a soma dos cheques. Na **Listagem 14** vemos o evento *OnItemEdit*.

Listagem 14. Editando um cheque selecinado

```
01 procedure TConsultaChequesF.ListViewItemEdit(pVT:
    TtiCustomVirtualTree;
02   AData: TtiObject; AItem: PVirtualNode);
03 begin
04  if EditChequesF.Execute(AData) then
05  begin
06    FListaCheques.Save;
07    ListView.Refresh(AData);
08    SomarTotais;
09  end;
10 end;
```

Como pode ser visto, o cheque selecionado é representado pelo parâmetro *AData* e da mesma forma após sua edição realizamos um *refresh* da exibição.

Para excluir um cheque selecionado utilizamos o evento *OnItemDelete*, que é exibido na **Listagem 15**.

Listagem 15. Excluindo um cheque

```
01 procedure TConsultaChequesF.ListViewItemDelete(pVT:
    TtiCustomVirtualTree;
02   AData: TtiObject; AItem: PVirtualNode);
03 begin
04   if tiPerObjAbsConfirmAndDelete(AData) then
05   begin
06     FListaCheques.Save;
07     pVT.Refresh;
08     SomarTotais;
09   end;
10 end;
```

Para solicitar ao usuário a confirmação da exclusão, utilizamos a função *tiPerObjAbsConfirmAndDelete*, que gera um diálogo e caso seja confirmada a exclusão, ele mesmo se encarrega de já excluir o objeto. Então apenas realizamos a atualização da exibição.

Gerando relatórios

Ainda não existe um gerador de relatório para *Delphi* que utilize, de forma simples, como fonte de dados uma lista de objetos. Todos geradores se utilizam de *TDatasets*. Pensando nisso, o *tiOPF* oferece um *DataSet* especializado que *transforma* objetos em registros, que podem ser ligados a controles *DBAware*. Esse *DataSet* é o *TtiDataSet*. Insira um no *DataModule dtm* e chame-o de *tiDataSetCheques*. Agora nesse *DataSet* podemos inserir as propriedades dos objetos que ele irá conter. Então adicionamos campos como mostra a **Figura 3**.

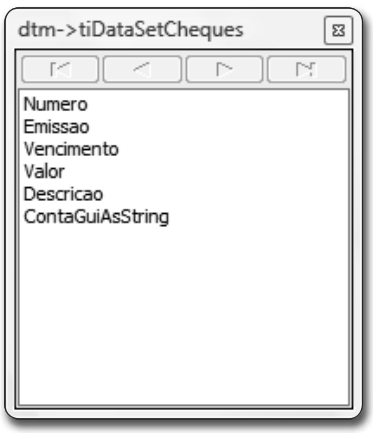


Figura 3. Campos disponíveis

Observe que esses campos possuem os mesmos nomes das propriedades da classe *TCheque*. Agora vamos criar um relatório no *QuickReport* como mostra a **Figura 4**.

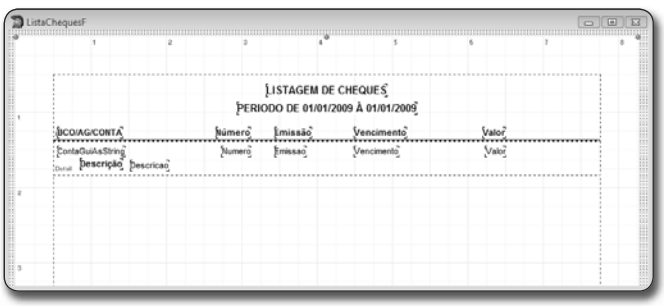


Figura 4. Relatório de cheques

Chamamos esse relatório de *ListaChequesF*. Adicionamos à sua seção *uses* uma referência ao *datamodule Dtm* e consequentemente ligamos os controles *DBAware* ao *tiDataSetCheques*, tudo como se fosse um relatório comum.

Nota

Não vou entrar em detalhes de como utilizar o QuickReport para geração de relatórios porque isso está fora do escopo do artigo.

Voltando ao formulário de consulta vamos codificar o botão *Imprimir* como mostra a **Listagem 16**.

Listagem 16. Imprimindo o resultado da consulta

```
01 procedure TConsultaChequesF.btImprimirClick(Sender: TObject);
02 begin
03   if FListaCheques.Count > 0 then
04   begin
05     dtm.tiDataSetCheques.LinkObject(FListaCheques, TCheque);
06     if dtm.tiDataSetCheques.Active then
07        dtm.tiDataSetCheques.Refresh
08     else
09        dtm.tiDataSetCheques.Open;
10
11     if rbEmissao.Checked then
12       ListaChequesF.qrTitulo.Caption := 'LISTAGEM DE CHEQUES
           EMITIDOS'
13     else
14       ListaChequesF.qrTitulo.Caption := 'LISTAGEM DE CHEQUES
           A VENCER';
15       ListaChequesF.qrPeriodo.Caption := 'NO PERÍODO DE ' +
           editData1.Text + ' À ' + editData2.Text;
16       ListaChequesF.QuickRep1.PreviewModal;
17     dtm.tiDataSetCheques.Close;
18   end;
19 end;
```

Antes de imprimir verificamos se existe algo a ser impresso verificando se existem itens em *FListCheques.* Caso seja verdade, observe como é simples transformar os objetos em registros de um *Dataset* através do método *LinkObject* do *tiDataSetCheques.*

Ao executarmos esse método, todo o conteúdo de *FListaCheques* é copiado para o *DataSet* , para o campos que definimos. Depois, apenas abrimos ou realizamos um *refresh* nesse *DataSet*. O restante é código comum para trabalhar com o *QuickReport*. Veja na **Figura 5** o resultado e uma consulta e na **Figura 6** a impressão desse resultado.

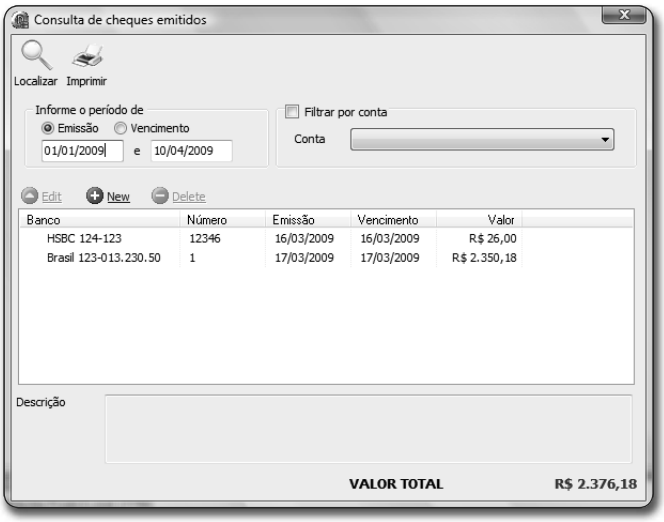


Figura 5. Resultado de busca

Figura 6. Impressão do resultado da busca

Conclusão

Como pode ser visto trabalhar com orientação a objetos no *Delphi* não é difícil, desde que você disponha das ferramentas certas. Ao utilizar o *tiOPF* ganhamos em produtividade pois não nos preocupamos com geração de *SQL* e até mesmo para construção de interfaces temos auxílio de seus componentes. Contudo utilizamos o recurso de *auto-mapping*, onde o próprio *tiOPF descobre* e gera as sentenças. Esse método é indicado para sistema pequenos, como o nosso. Para sistemas grandes a própria equipe do *tiOPF* sugere um outro recurso de mapeamento chamado *HardCoded SQL*, onde você mesmo gera as sentenças *SQL*, procurando otimizá-las e elas serão utilizadas pelo *framework*. Mas, isso é assunto para um próximo artigo. Espero que o *tiOPF* desperte o interesse pela aplicação da orientação a objetos em seus sistemas. Abraço. ●

Dê seu feedback sobre esta edição!

A Clubedelphi tem que ser feita ao seu gosto. Para isso, precisamos saber o que você, leitor, acha da revista!

Dê seu voto sobre este artigo, através do link:

www.devmedia.com.br/clubedelphi/feedback

Existem coisas que não conseguimos ficar sem!

...só pra lembrar, sua assinatura está acabando!

Renove Já!

www.devmedia.com.br/renovacao

Nesta seção você encontra artigos intermediários sobre Delphi Win 32 e Delphi for .NET

Conhecendo o RapidSQL

Como gerenciar bases de dados com o RapidSQL?

Nesse artigo veremos
- Download e instalação do RapidSQL;
- O que é o RapidSQL;
- Como utilizá-lo na prática.

Qual a finalidade
Apresentar a ferramenta e mostra como utilizá-la no dia-a-dia para manutenção de bancos de dados;

Quais situações utilizam esse recurso?
A todo momento a ferramenta pode ser utilizada. Tanto para a criação quanto manutenção de bancos de dados é recomendado que se use alguma ferramenta front-end;

Adriano Santos
falecom@adrianosantos.pro.br
É desenvolvedor Delphi desde 1998, professor e programador PHP, bacharel em Comunicação Social pela Universidade Cruzeiro do Sul, SP. É Editor Chefe das revistas ClubeDelphi e WebMobile. Gerente de TI na SoftPark, parceira Borland, membro fundador do DUG-SP (www.dug-sp.com) e mantenedor do blog Delphi to Delphi (www.delphitodelphi.blogspot.com) com dicas, informações e tudo sobre o desenvolvimento Delphi.

Resumo do DevMan

Trabalhar com desenvolvimento no dia-a-dia pode se tornar uma batalha quando não se tem as ferramentas corretas a mão ainda mais se tratando de banco de dados. Quem está acostumado a desenvolver sem um DBA ao lado, o que é a grande maioria dos desenvolvedores no Brasil, sabe que mexer com SQL não é fácil nem totalmente amigável. Por isso diversas ferramentas surgem todos os dias no mercado, como é o caso o IBExpert , IBOConsole, SQL Studio Manager, entre outros.

Nesse artigo quero apresentar uma ferramenta que gostei bastante e é cheia de recursos. O RapidSQL é uma ferramenta paga desenvolvida pela Embarcadero Technologies, atual proprietária dos compiladores JBuilder, Delphi, C++ Builder, Delphi for PHP e também do banco de dados Interbase.

Testei pessoalmente a ferramenta e digo que vale a pena o investimento.

Estive participando de um treinamento da *Embarcadero Technologies* no Brasil, recentemente, onde fui apresentado aos seus principais produtos da empresa. Confesso que me surpreendi com a gama de produtos e serviços que eles disponibilizam e particularmente com a qualidade dos softwares. Um dos aplicativos mais interessantes do treinamento foi o *RapidSQL*. Ele tem exatamente o

mesmo intuito de outros programas concorrentes, tais como *EMS SQL Studio Manager* e *IBExpert*.

O objetivo do artigo é dar mais uma opção ao desenvolvedor que desejar adquirir uma ferramenta para manipular e prestar manutenção em seus bancos de dados. Veremos nesse artigo como baixar e instalar, bem como algumas de suas principais características. Mão na massa.

Baixando e instalando o Trial

Para que possamos conhecer de perto o aplicativo, vamos baixar e instalar uma versão em nosso sistema e brincar um pouco com ela. A *Embarcadero Technologies* disponibilizou para download em regime *Trial* (30 dias) o software *RapidSQL* em seu site no endereço *www.embarcadero.com*. Para efetuar o download da ferramenta é necessário acessar o menu *Products* e em seguida *RapidSQL*. No banner principal clique no botão *Download Trial* e então você será redirecionado para uma nova página. É importante que você tenha cadastro no site. Se já tinha quando era *CodeGear*, então não se preocupe porque o cadastro continua valendo. Se não tiver, basta se cadastrar gratuitamente.

Bem, faça o download da primeira opção *Rapid SQL 7.5.5 Professional Edion*. Faremos nossos testes nessa versão do aplicativo. Após efetuar o download de cerca de *160Mb* de tamanho, execute o arquivo *rs_755_3386_trial.exe*. Uma tela de boas vindas como na **Figura 1** será exibida.

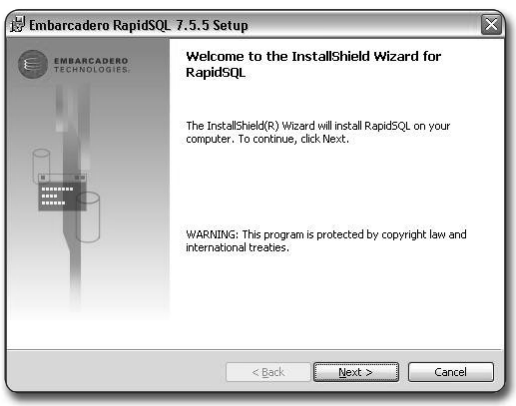


Figura 1. Tela de boas vindas do RapidSQL

A instalação é bem suave e não requer nenhuma configuração específica. Apenas continue selecionando *Next* nas telas seguintes. Ao ser indagado do tipo de instalação, como na **Figura 2**, selecione a opção *Workstation Installation*.

Utilizando o Rapid SQL

Instalado o aplicativo abra-o selecionando o menu *Iniciar>Embarcadero Rapid SQL 7.5.5*. O *splash* (**Figura 3**) de abertura será mostrada logo de início uma mensagem indagando se deseja ou não registrar os *DataSources* para os bancos já instalados no computador (**Figura 4**). Os *DataSources* são identificados automaticamente pelo sistema, ou seja, caso o *Oracle, SQL Server* e outros bancos nativos estejam instalados e rodando no computador, o *RapidSQL* os identificará.

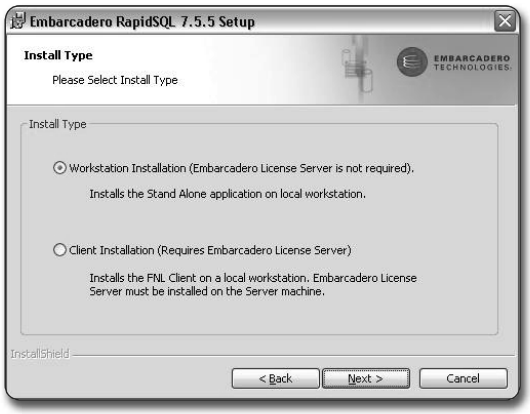


Figura 2. Tipo de instalação do Rapid SQL

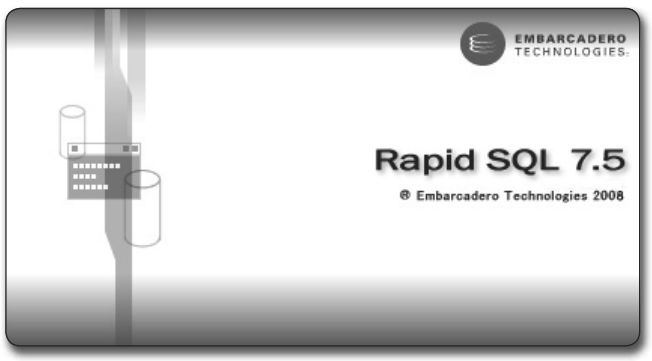


Figura 3. Splash de abertura

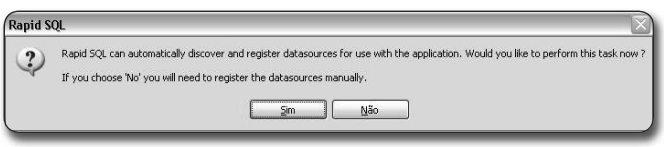


Figura 4. Solicitação de registro de DataSources

A IDE do programa já pode ser vista (**Figura 5**) e no centro dela fica disposta uma caixa de diálogo. Clique em Ok para que possamos criar ou registrar uma base de dados em nosso programa. Clicando em Ok, percebemos que nativamente o programa possui acesso direto aos bancos *IBM DB2, Microsoft SQL Server, MySQL, Oracle* e *SyBase*. Demais bases de dados tais como *Interbase, Firebird e PostGreeSQL* não possuem acesso direto, somente via *ODBC*, ou seja, é necessário selecionar a último opção dessa tela para tais bancos. Segundo membros representantes da *Embarcadero*, os desenvolvedores trabalham na compatibilidade do *Rapid SQL* com o *Interbase*, agora também produto da empresa.

Farei o teste especialmente com o *MySQL 5.06* que tenho instalado em minha máquina virtual de testes. Só para que possamos entender melhor, o *RapidSQL* se adapta as necessidades de cada banco de dados e tipo de conexão. Por exemplo, experimente apenas selecionar o *Microsoft SQL Server*, clicar em avançar. Agora retorne, selecione *Oracle* e avance. Perceba que as informações de credenciais (**Figuras 6** e **7**) solicitadas são diferentes de acordo com o banco selecionado. Em outras palavras o *Rapid SQL* se adapta ao banco de dados atual, mostrando ou ocultando recursos. Isso contempla não somente no *login*, mas em toda a extensão do produto.

Figura 5. Interface do programa

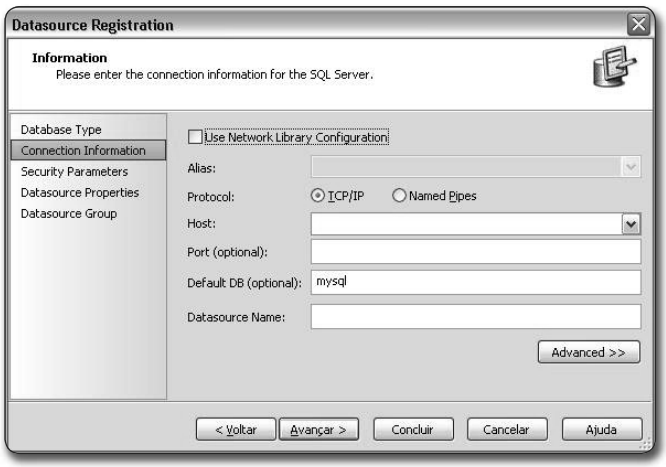


Figura 6. Dados para conexão com MySQL

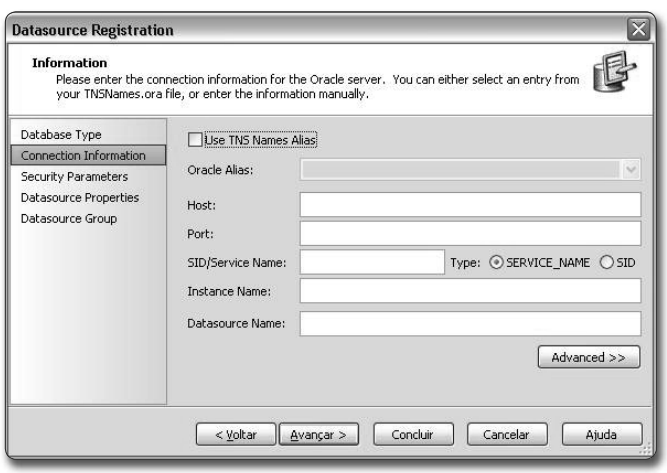


Figura 7. Dados para uso com Microsoft

Após ter configurado uma conexão com um ou mais servidores, estamos prontos para registrar as bases de nossos para efetuamos os primeiros testes. Veja que a esquerda do programa encontra-se uma árvore de itens. Essa área é destinada ao banco de dados (**Figura 8**). Nela teremos todas as bases que necessitarmos, assim como já fazemos com outros programas equivalentes.

Podemos notar que a parte superior do programa é repleta de botões com muitas funções. São ao todo quatro barras conforme

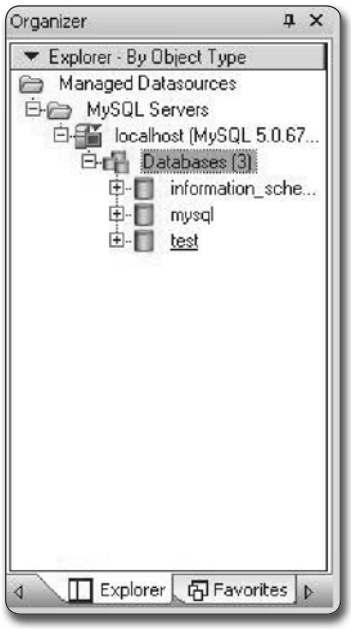


Figura 8. Servidor MySQL instalado

vemos na **Figura 9**. De cima para baixo e da esquerda para a direita, a primeira barra possui os atalhos específicos para tratar o *host*. Os três primeiros servem para *Registrar, Editar* e *Excluir Host*. Em seguida vemos um *ComboBox* que lista todos os *host's* possíveis. Podemos registrar mais de um *host* para cada tipo de base de dados, por exemplo, na minha máquina virtual registrei dois servidores para o *MySQL* como podemos ver na **Figura 10**. Não é possível notar, mas os dois hosts do *MySQL* estão apontando para o mesmo endereço, *localhost*. Mas eu poderia muito bem ter apontado um servidor remoto e outro local.

Após a lista de hosts, temos a lista de *DataBases* do servidor atual e alguns ícones referentes a organização de janelas. A segunda barra contém atalhos para criação de tabelas, *scripts*, campos etc. Além dos botões para *Salvar*, abrir o banco e coisas do gênero. As demais barras possuem atalhos para execução de *scripts*, *commits*, *rollbacks*, etc.

Registrando o host e criando bancos e tabelas

Apesar de todos esses atalhos, isso acaba sendo comum em qualquer programa dessa natureza. Por isso vamos ver o que realmente nos interessa. A primeira coisa a se fazer é registrar um *Host*. Para isso clique com o botão direito do mouse sobre o item *Managed Datasources* a esquerda do sistema e em seguida em *Register Datasource*. Na tela que se abre, selecione *MySQL* ou o banco de dados que desejar. Clique em avançar e preencha todos os dados requeridos, tais como endereço do *host*, usuário e senha e confirme.

Depois de criado o *Host* uma pasta será criada, no meu caso ficou como *MySQL Servers*. Evidentemente que é possível renomear a pasta para algo mais conveniente. Para criar o banco de dados agora, é necessário clicar com o direito em *DataBases>New*. Uma nova janela é aberta (**Figura 11**), apenas preencha com os dados necessários e siga o tutorial. Ao final, perceba que o *Rapid SQL* mostra o *script* que será executado para criação do banco (**Figura 12**).

Figura 9. Barras de ferramentas

Figura 10. Hosts para o MySQL

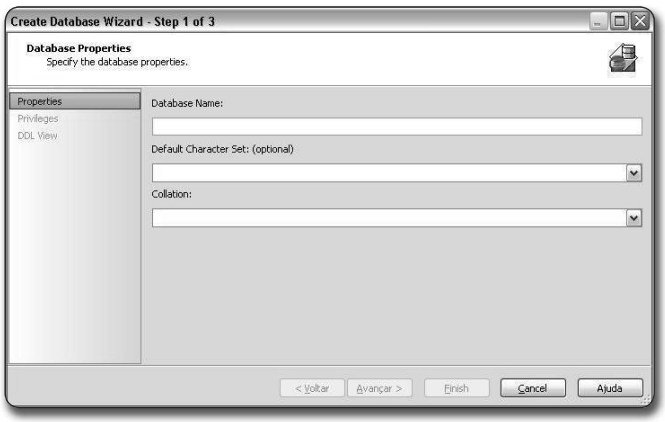


Figura 11. Configuração do servidor

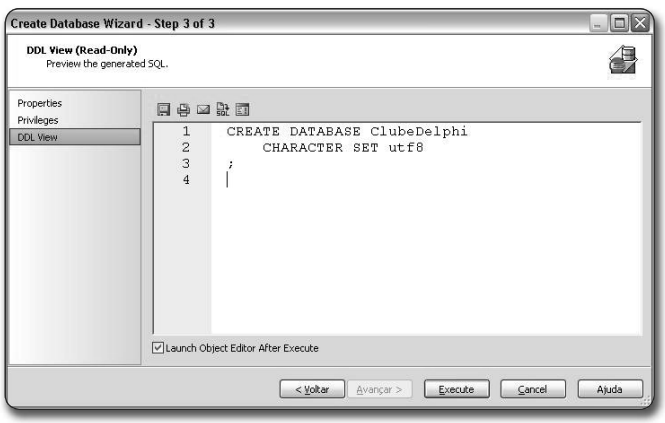


Figura 12. Script para criação do banco

Uma das principais características das aplicações da *Embarcadero* é manter o usuário informado de tudo o que está acontecendo no momento. Por isso, grande parte das operações é notificada em tela.

Você notará que à esquerda do sistema temos o BD criado anteriormente e os objetos sendo mostrados. Vamos criar agora uma nova tabela. Selecione o item *Table* e clique com o botão direito nele usando a opção *New*. Na janela que se abre, digite o nome da tabela no campo contornado de vermelho. Em seguida clique na opção *Columns* a esquerda da caixa de diálogo. Clique no botão *Add column* e adicione quantas colunas desejar. No meu caso inclui apenas *ID* e *Cliente*. Por fim clique em *Finish* e confira o *script* gerado pelo sistema. Seguindo os mesmos passos crie uma nova tabela com o nome *Pedidos*. Ela deverá conter os campos como a seguir:

- ID Int Not Null;
- ID_CLIENTE Int Not Null;
- DATA Time Stamp;
- VALOR Float Not Null;

A quantidade de ferramentas e comandos é impressionante. Experimente clicar com o botão direito na tabela criada. Você verá um menu bastante completo para auxiliar ainda mais o desenvolvedor no dia-a-dia (**Figura 13**).

Vejamos por exemplo como fazer uso do *Query Builder*. Há um menu destinado somente ao recurso mencionado. Clique no meu *Tools>Query Builder* para abrir a janela. Na parte esquerda são mostradas todas as tabelas do sistema, agora basta arrastá-las para o espaço branco a direita e fazer as devidas ligações e marcações dos campos desejados.

Arraste as duas únicas tabelas que temos no banco de dados, *Cliente* e *Pedidos*. Em seguida clique no campo *ID* e arraste-o para

Figura 13. Ferramentas do RapidSQL

cima do campo *ID_CLIENTE* na tabela *Pedidos*. Por fim marque todos os campos de ambas tabelas. Teremos como resultado um cenário semelhante ao visualizado na **Figura 14**. Agora basta clicar no botão *Play* para que a *query* seja mostrada.

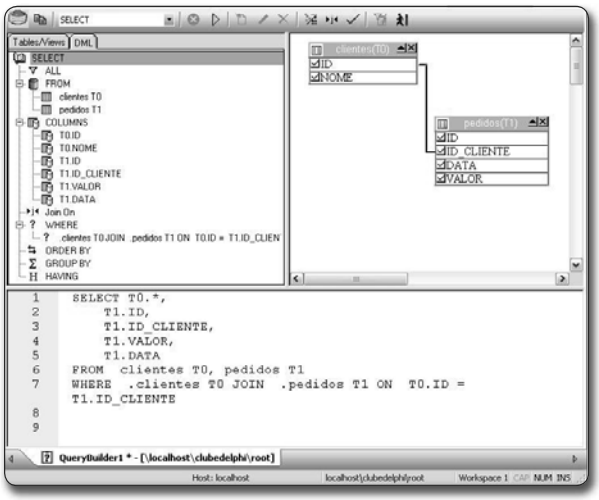


Figura 14. Query Builder em ação

Inserindo dados na tabela

Como todos *front-end* para bancos de dados, é possível incluir registros diretamente nas tabelas. Isso é feito selecionando o item *Edit Data* ao clicar com o botão direito do mouse sobre a tabela. Você irá notar que o programa abre uma caixa de diálogo. Nela você informará quais campos serão usados na inserção, assim evita-se de ter todos os campos na tela sendo que vamos inserir dados em apenas alguns.

Na parte superior da grade que é aberta, temos alguns botões típicos como os navegadores de registro e o botão para aplicação de filtros. O interessante dessa tela é que ele guarda e nos mostra todos os passos efetuados durante a manutenção da tabela. Veja que o código *SQL* foi gerado automaticamente pelo sistema e mostrado em uma janela na parte inferior (**Figura 15**).

Outros programas, como o *IBExpert*, guardam o histórico, mas não o mostram na tela como o *RapidSQL*. A informação é importante, pois com isso podemos gerar scripts para a atualização de bancos de dados de clientes por exemplo.

Visual Diff – Comparando Scripts

Para quem utiliza constantemente *scripts* para atualização, controlar as alterações deles é difícil. Pior ainda quando o script é grande e precisamos comparar versões diferentes. O *RapidSQL* possui um recurso chamado *Visual Diff*. Com ele podemos informar dois arquivos a serem comparados e ele nos mostra onde estão as diferenças. Para isso basta acionar o menu *Tools>Visual Diff*. E uma tela dividida ao meio aparecerá. A esquerda clique no primeiro botão e escolha o *script* 1, em seguida clique no segundo botão e escolha o *script* 2. Na pasta que abre por default existem alguns arquivos de exemplo do próprio *Rapid*, para não perder tempo vamos direto selecionar os arquivos *SybIdxSize119.sql* e *SybIdxSize125.sql*. Após selecionar o

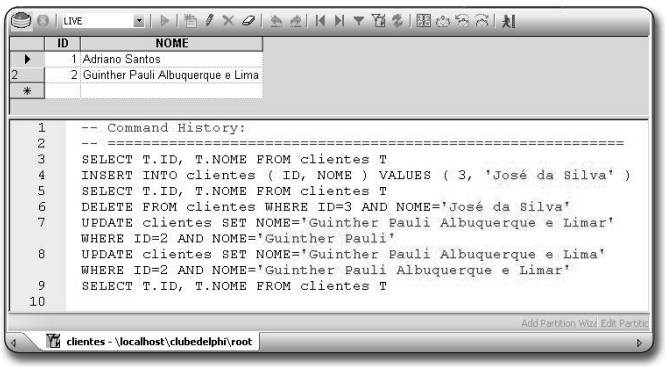


Figura 15. Histórico SQL na janela de manutenção

segundo botão, o sistema já fará a varredura nos dois arquivos e apontará onde estão as diferenças (**Figura 16**).

Role a tela para baixo e perceba que há algumas cores no texto agora. Elas representam as mudanças, veja a legenda:

- *Azul:* texto alterado;
- *Vermelho:* texto inserido existente na versão antiga;
- *Verde:* texto inserido na versão nova;

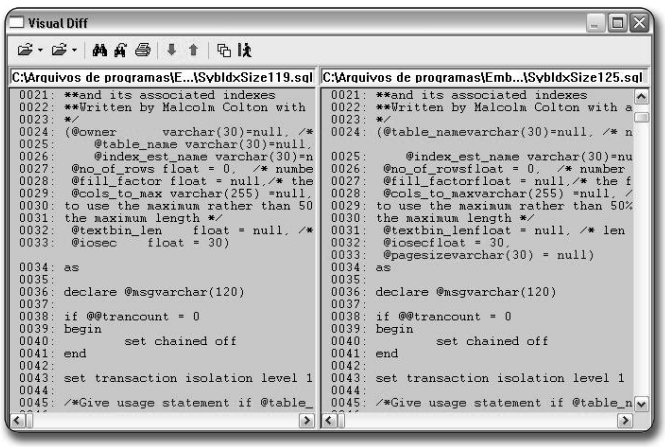


Figura 16. Visual Diff

Acessando o servidor Firebird via ODBC

Como mencionado anteriormente, o *RapidSQL* não possui acesso nativo a todos os bancos, por isso é importante saber fazer conexão utilizando um *driver ODBC*. Praticamente todos os bancos de dados possuem um *ODBC driver*, então basta que façamos a instalação dele no computador que termos acesso facilmente ao *BD*. Vamos fazer a configuração para manipulação de bancos de dados do Firebird.

A primeira providência é baixar um *driver ODBC* então. Vamos ao site do *Firebird* em *www.firebirdsql.org* no link *Download*. Apenas pare o mouse em *Download* e note que há uma opção *Firebird ODBC driver*, clique nela. Na página encontraremos diversos arquivos para baixar. Baixe a que melhor se encaixe com seu perfil, no meu caso eu baixei o *Windows 32-bit Full Install Firebird ODBC 2.0.0.148*. Um arquivo com o nome *Firebird_ODBC_2.0.0.148_win32.exe* foi copiado para o meu computador, então basta instalar o *driver* seguindo o fluxo do assistente e voltar para o *RapidSQL*, para que possamos incluir um *host* para o *FB*.

Nota do DevMan

ODBC (acrônimo para Open Data Base Connectivity) é um padrão para acesso a sistemas gerenciadores de bancos de dados. Este padrão define um conjunto de interfaces que permitem o uso de linguagens de programação como Visual Basic, Delphi, Visual C++, entre outras capazes de utilizar estas interfaces, para ter acesso a uma vasta gama de bases de dados distintas sem a necessidade de codificar métodos de acesso especializados.

Retorne ao aplicativo e clique em *Tools>ODBC Administrator*. Na janela que se abre clique em *Fonte de Dados de Sistema* e em seguida em Adicionar. Localize *Firebird/Interbase(r) driver* (**Figura 17**) e clique em *Concluir*. Agora basta preencher todos os dados necessários para fazer a conexão com o banco. O primeiro campo (*Data Source Name*) é o nome do nosso banco de dados. Coloquei como *Employee*, que é o arquivo de exemplo do próprio *FB*. Na seqüência é necessário selecionar o arquivo do banco propriamente dito, que por *default* fica armazenado em *C:\Arquivos de programas\Firebird\Firebird_2_1\examples\empbuild\EMPLOYEE.FDB*. Também é necessário informar o caminho completo para a *DLL client* (*fbclient.dll*) presente em *C:\Arquivos de programas\Firebird\Firebird_2_1\bin\fbclient.dll*. Por fim entre com usuário e senha *(SYSDBA, masterkey)* e o *CharSet default (WIN1252)* (**Figura 18**). Se preferir, clique em *Test Connection* para testar se a configuração foi bem sucedida.

O próximo passo agora é acrescentar um host apontando para esse banco do *Firebird*. Clique com o direito em *Managed Data-Sources* e em seguida em *Registration DataSource*, escolha *ODBC* e avance. Basta agora selecionar *Employee* no *ComboBox* referente a *Connection String* e avançar. Na tela seguinte preencha com as credenciais do *FB* e conclua. Uma nova pasta e o banco de dados foram adicionados na árvore a esquerda (**Figura 19**).

Essa última imagem, note que registrei em meu servidor bancos *Oracle XE, SQL Server 2005 Express, MySQL* e *Firebird*. Em outras palavras, posso trabalhar tranquilamente com apenas uma ferramenta instalada e fazendo manutenção em diversos bancos de dados, o que acarreta em redução de custos.

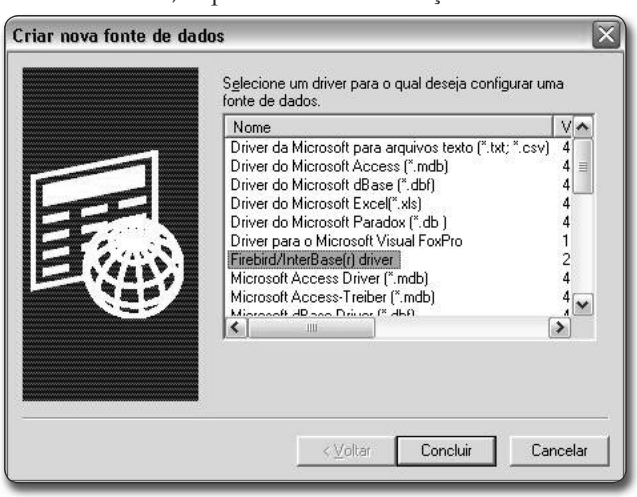


Figura 17. Selecionando o driver ODBC

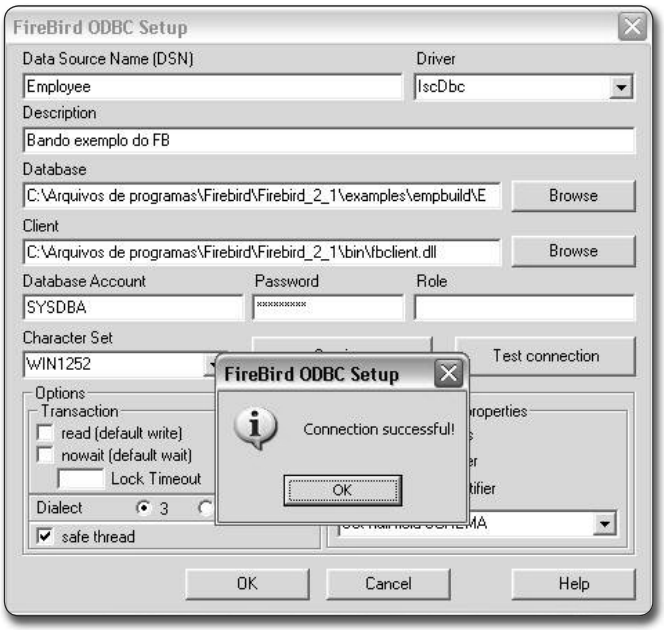


Figura 18. Entrando com os dados de conexão

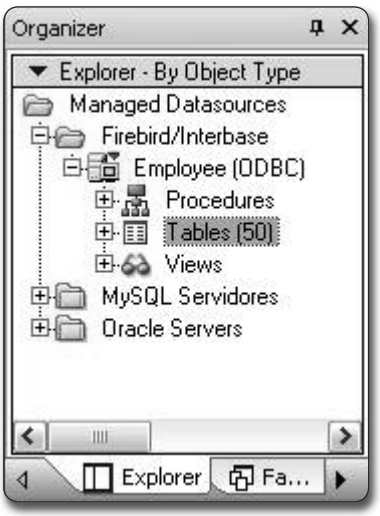


Figura 19. Firebird adicionado

Conclusão

Reduzir custos e ter a praticidade de trabalhar em apenas uma ferramenta de bancos de dados, talvez não tenha preço. O importante de tudo isso é saber que há a possibilidade de adquirirmos uma ferramenta robusta, rápida e que nos atende no dia-a-dia sem nenhum problema.

No site da *Embarcadero* existem outras ferramentas interessantes e que vale a pena conferir de perto. Um forte abraço e até a próxima. ●

Dê seu feedback sobre esta edição!

A Clubedelphi tem que ser feita ao seu gosto. Para isso, precisamos saber o que você, leitor, acha da revista!

Dê seu voto sobre este artigo, através do link:

www.devmedia.com.br/clubedelphi/feedback

Nesta seção você encontra artigos para iniciantes na linguagem Delphi

Métodos da teoria à prática

Entenda de forma simples e prática como trabalhar com métodos em Delphi

Nesse artigo veremos

- Como funcionam os métodos de forma prática.
- Maneiras de definí-los, bem como a compreensão da necessidade.
- Exemplo prático, e implementação de uma biblioteca de métodos.

Qual a finalidade

Tornar clara e compreensível o entendimento, a utilidade e a problemática para defini-los, agregando maior organização e qualidade ao seu software.

Quais situações utilizam esse recurso?

Qualquer software pode possuir estes recursos, tornando mais produtivo o desenvolvimento e conseqüentemente facilitando sua manutenção.

Resumo do DevMan

É comum a nós desenvolvedores em nosso ambiente de trabalho ou até mesmo fora dele realizar manutenções e adaptações em softwares antigos, que muitas vezes não tiveram um planejamento tão eficaz, ou não foram implementados de maneiras mais práticas, possuindo códigos repetitivos e mal-organizados. A criação de métodos e rotinas genéricas podem fazer com que o desenvolvimento se torne mais fácil em questões de manutenção, entendimento e é claro, organização.

Daniel Sobrinho Laporte
daniel.laporte@gmail.com
É graduado em sistemas de informação e pós-graduado em administração de bancos de dados pela Universidade de Franca. É analista de sistemas e desenvolvedor Delphi desde 2005 desenvolvendo aplicações cliente/servidor e multicamadas em Delphi com Firebird.

Quando estamos iniciando o aprendizado de uma linguagem de programação, é comum que surjam muitas dúvidas relacionadas à maneira de como resolveremos nossos desafios da melhor maneira possível. Muitas vezes temos o conhecimento teórico necessário, porém, por falta de experiência, ou por outros fatores, acabamos não adotando estas teorias em nossos softwares.

É comum ao trabalharmos com manutenção de software, nos depararmos com códigos repetitivos como validações simples, dezenas de cálculos, ou até mesmo rotinas implementadas de forma errada ou desnecessária. Neste artigo, o nosso objetivo é fazer o melhor entendimento

possível destas situações (teóricas) para que assim, possamos minimizar nosso retrabalho, tornando nossos softwares mais produtivos, com maior qualidade e é claro, tornar o nosso trabalho cada vez mais simples, ágil e eficiente.

Para isso, começaremos entendendo conceitos simples, para que possamos seguir para a prática, e aumentar a dificuldade gradativamente, no entanto, é necessário antes entendermos um dos principais conceitos que envolvem nosso trabalho. A análise.

Entendendo a problemática

Quando vamos iniciar o desenvolvimento de qualquer software, tudo tem início por meio de uma análise prévia que identifique as necessidades do desenvolvimento. Essa etapa é denominada levantamento de requisitos, ou seja, buscar todas as situações, extrair todas as tarefas que nosso produto se encarregará de fazer, para assim dar continuidade às demais etapas que seguem o desenvolvimento.

Em se tratando do desenvolvimento não é diferente. Para implementarmos boas práticas de programação, é sempre necessário que se realize a análise do que se quer fazer, pensando sempre no que iremos precisar para determinar nosso métodos, se iremos reaproveitá-los em outras partes do software criando-os da maneira mais funcional possível, realizando assim um pequeno planejamento.

Conceitos iniciais

Basicamente iniciaremos explicando a diferença entre as *Procedures* e *Functions*. Estes conceitos são denominados métodos, e são comuns a todas as linguagens diferenciando-se apenas pela sua síntaxe (forma como é declarada).

As *Procedures* como o próprio nome sugere, são procedimentos que realizam determinada ação, porém, são caracterizadas por realizar esta tarefa a si designada e não retornar um valor.

Ao contrário das *Procedures*, as *Functions* realizam uma determinada ação, no entanto, retornam um valor de acordo com a nossa necessidade, e logicamente baseado em nossa análise.

Bem vamos deixar um pouco da teoria de lado e partir para a prática para exibir a implementação simples destes exemplos iniciais.

Iniciando o desenvolvimento

Vamos iniciar o nosso projeto de exemplo, abra o seu *Rad Studio 2007* e crie uma nova aplicação (*File > New > VCL Forms Application – Delphi for Win32),* inicie alterando as propriedades *Name* (*frmPrincipal*) e *Caption (Exemplo de Métodos),* em seguida adicione um componente *GroupBox* e altere seu *Caption* para *Procedure de Soma* , adicione três *Edit's* (*edtProc1*, *edtProc2*, *edtProc3*) e um *Button* (*btnProcSomar*), adicione dois *Label's* e deixe seu *Form* com a aparência semelhante ao da **Figura 1.** Salve seu projeto nomeando a *unit* como *untPrincipal.pas* e o projeto como *Metodos.dpr.*

Agora iremos dar inicio a codificação, fazendo uma procedure de soma, para exemplificarmos a sua utilização.

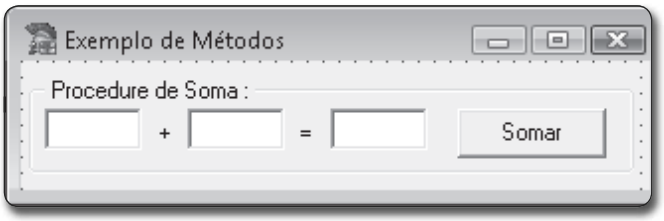


Figura 1. Formulário de exemplo

Digite o código da listagem a seguir para que possamos criar nosso procedimento, em seguida pressione simultaneamente as teclas *Ctrl + Shift + C* para que o cabeçalho do método seja declarado automaticamente.

```
procedure TfrmExemplo.ProcSoma(Valor1, Valor2: Integer);
begin
  edtProc3.Text := IntToStr(Valor1 + Valor2);
end;
```

Repare que apenas declaramos um procedimento que receberá dois parâmetros de tipo *Integer*, realizando a soma dos valores e convertendo-os em um tipo *String* para que tenham o conteúdo atribuídos ao *Edit edtProc3.*

Em seguida, dê um duplo clique no botão *btnProcSoma* e implemente o evento *OnClick* conforme o trecho de código adiante:

```
ProcSoma(StrToInt(edtProc1.Text), StrToInt(edtProc2.Text));
```

Observe que os valores passados como parâmetros para a nossa *Procedure,* referem-se ao conteúdo digitado nos *Edit's (edtProc1* e *edtProc2),* mas caso estes valores não estejam preenchidos, a nossa *procedure* retornará um erro, pois não será possível realizar a conversão de uma *String* vazia para um número inteiro.

Assim vamos criar um método que funcionará como uma pergunta, questionando se o conteúdo digitado está vazio ou não, e retornando assim a confirmação que necessitamos para realizar este procedimento. Para isso digite o trecho de código adiante, e novamente pressione simultaneamente as teclas *Ctrl + Shift + C* para que o cabeçalho da nossa rotina seja criado.

```
function TfrmPrincipal.TextoVazio(Texto: String): Boolean;
begin
  Result := (Trim(Texto) = '');
end;
```

Como havíamos comentado antes, uma *Function* implementa uma ação e retorna um resultado através da cláusula *Result*. Neste caso, nossa função recebe como parâmetro o texto digitado e retornando *False* caso o conteúdo do parâmetro esteja vazio e *True* caso contrário. Sendo assim, modificaremos o código do evento *OnClick* do *btnProcSoma* para o da listagem adiante:

```
if(not TextoVazio(edtProc1.Text)) and (not TextoVazio(edtProc2.
Text))then
  ProcSoma(StrToInt(edtProc1.Text), StrToInt(edtProc2.Text));
```

Executando sua aplicação através do botão *Run (F9),* será possível observar que tudo está certo, porém, se adicionarmos mais um componente *Edit* para que o resultado seja atribuído aos dois simultaneamente, você irá reparar que essa solução

terá que ser adaptada para que o novo componente também receba o valor.

Deste modo, após analisarmos esta *Procedure*, não seria melhor se implementássemos algo que nos retornasse o valor já somado para que pudéssemos atribuir ao componente que quiséssemos, e em qualquer situação que necessite de uma soma entre números inteiros? Pois bem, faremos essa nova alteração. Redimensione o seu formulário e adicione mais um componente *GroupBox* alterando seu *Caption* para *Função de Soma*, juntamente com outros três *Edit's*(*edtFunc1*, *edtFunc2*, *edtFunc3*) e um *Button(btnFuncSomar)*, adicione dois *Label's* deixando seu *Form* com a aparência semelhante ao da **Figura 2.**

Figura 2. Formulário de exemplo após alterações

Agora iremos codificar a rotina responsável por realizar a soma de uma forma genérica, podendo assim ser reutilizada a qualquer momento dentro do formulário, de forma que realize o cálculo baseado nos valores inteiros e retorne apenas o seu resultado. Digite assim o código a seguir e crie novamente o cabeçalho referente à função:

```
function TfrmPrincipal.FuncSoma(Valor1, Valor2: Integer): Integer;
begin
  Result := Valor1 + Valor2;
end;
```

Desta forma, iremos fazer com que ao clicar no botão *btnFuncSomar*, o resultado da soma seja atribuído ao *edtFunc3*, lembrando que com esta rotina que acabamos de implementar é possível realizarmos o cálculo atribuindo a qualquer componente, portanto digite o código da listagem seguinte no evento *OnClick* do botão e em seguida faça o teste executando o programa.

```
if(not TextoVazio(edtFunc1.Text)) and (not TextoVazio(edtFunc2.
Text))then
  edtFunc3.Text := IntToStr(FuncSoma(StrToInt(edtFunc1.Text),
StrToInt(edtFunc2.Text)));
```

Na listagem anterior é possível observamos que apenas é validado o conteúdo dos *Edit*, e caso haja algo preenchido a função *FuncSoma* retornará o valor ao componente que definimos. Agora que entendemos um pouco mais de métodos, vamos nos familiarizar e entender como funciona um conceito que utilizamos a todo momento. A definição de propriedades.

Utilizando Propriedades

Trabalhando na *IDE(Integrated Development Environment)* do *Delphi*, nos deparamos com este conceito o tempo todo. Ele está

Nota do DevMan

A sigla IDE, do inglês Integrated Development Environment ou Ambiente Integrado de Desenvolvimento, é um programa de computador que reúne características e ferramentas de apoio ao desenvolvimento de software com o objetivo de agilizar este processo. (Fonte: Wikipédia)

presente em cada objeto, formulário ou componente criado, sendo facilmente visualizado no *Object Inspector* conforme ilustra a **Figura 3.**

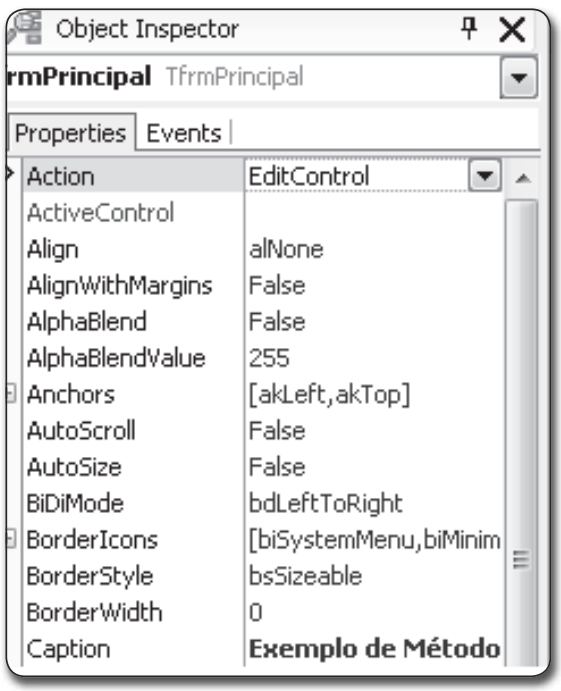


Figura 3. Propriedades no Object Inspector do Delphi

Para entendermos o seu funcionamento, iremos criar uma propriedade em nosso formulário atual, e exibir uma aplicação de seu uso. Para isso prepararemos o nosso formulário para receber estes conceitos. Adicione assim um componente *GroupBox* e altere seu *Caption* para *Propriedades*, dento deste, insira dois *Button's (btnOculta e btnExibe)*, após realizar estas modificações verá que a nova aparência será semelhante ao da **Figura 4.**

Vamos agora criar nosso exemplo de propriedade. Siga à sessão *public* do seu formulário e digite sua declaração de acordo com o código da listagem a seguir:

```
property VisibleButton : Boolean read FVisibleButton write
SetVisibleButton;
```

Pressionando *Ctrl* + *Shift* + *C*, observaremos que o *Delphi* encarrega-se de criar automaticamente no escopo *private* do formulário uma variável denominada de *FVisibleButton* do tipo *Boolean*, e uma *procedure* identificada como *SetVisibleButton*. Podemos atentar que o corpo desta *procedure* já é instantaneamente criado com o seguinte código:

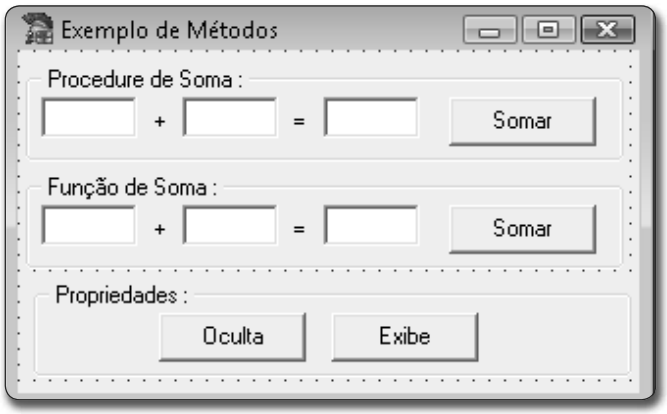


Figura 4. Preparação do formulário para a aplicação das propriedades

```
FVisibleButton := Value;
```

Aumentando um pouco a complexidade, vamos entrar em um fundamento muito visto em conceitos de *POO (Programação Orientada a Objeto)*, onde temos a utilização de dois métodos que são comumente denominados de *Get* e *Set*, fazendo com que o método *Get* retorne algo e o método *Set* atribua algo. Neste caso não é diferente, o que o *Delphi* fez para nós, foi criar uma variável que receberá um valor *Booleano* simulando o método *Get*, e definir uma atribuição à mesma por meio da *procedure*, simulando assim um método *Set*. Portanto, podemos buscar o valor atual do método *Get* acessando esta variável, ou realizar o processo de atribuição através da *procedure*. Agora vamos aplicar esta propriedade as nossas necessidades.

Primeiramente vamos fazer uma pequena modificação na *procedure SetVisibleButton*, codificando-a como a listagem que se segue:

```
FVisibleButton := Value;
btnOculta.Visible := FVisibleButton;
btnExibe.Visible  := not FVisibleButton;
```

Como o valor do parâmetro (definido pela constante *Value)* é do mesmo tipo da nossa variável criada pelo *Delphi* teremos a atribuição do valor definido a nossa variável *(Set)*, em seguida, atribuiremos os valores de *FVisibleButton* à propriedade *Visible* do componente *btnOculta* e a situação inversa ao componente *btnExibe*, deste modo apenas um dos botões estará visível por vez.

Faremos agora uma adaptação no nosso formulário, pois assim como as propriedades do *Object Inspector* são definidas com um valor *default*, queremos que seja atribuído um valor padrão à nossa propriedade, dessa forma adicione o seguinte código no evento *OnShow* do formulário:

```
VisibleButton := True;
```

De acordo com a nossa propriedade, o valor *True* fará com que o botão *btnOculta* esteja visível, e como a visibilidade do botão *btnExibe* é a invertida através do operador *not*, teremos o resultado *False* para este, fazendo com que fique invisível. A próxima listagem deve ser adicionada no evento *OnClick* do *btnOculta* para que o outro botão seja ocultado.

```
VisibleButton := False;
```

Para o evento *OnClick* do *btnExibe* simplesmente faça a codificação conforme já visto no evento *OnShow* como mostrado a seguir:

```
VisibleButton := True;
```

Executando o programa você perceberá que o botão *Exibe* não está visível inicialmente, porém ao clicar no botão *Oculta*, este será automaticamente ocultado e exibindo o outro botão.

Nota

É perfeitamente possível criar propriedades do tipo que você desejar, ou necessitar, sendo de valores comuns como String, Integer, Boolean, ou até mesmo de objetos como TDataSet, TClientDataSet por exemplo.

Agora que compreendemos os conceitos de métodos e a aplicação de propriedades iremos definir uma classe de funções para o nosso sistema. Veremos adiante quais as vantagens de utilizá-la.

Acesse o menu *File>New>Unit – Delphi for Win32* para criarmos uma nova *Unit*, salve seu projeto e defina seu nome como *untFunctions.pas* e faça sua implementação como mostra a **Listagem 1.**

Listagem 1. Implementação da Unit untFunctions

```
01  unit untFunctions;

02  interface

03  uses Variants;

04    type
05      TOperacao = (toSoma, toDivisao, toSubtracao, toMultiplicacao);
06      TMathClass = class
07      private
08        class function Divisao(Valor1, Valor2 : Integer) : Double;
09        class function Multiplicacao(Valor1, Valor2 : Integer) :
            Integer;
10        class function Subtracao(Valor1 , Valor2 : Integer) :
            Integer;
11        class function Soma(Valor1, Valor2 : Integer) :
            Integer;
12      public
13        class function ExecutaOperacao(Valor1, Valor2 :
            Variant; Operacao : TOperacao) : Variant;
14    end;
15  implementation
```

Perceba que na linha 05 foi definido um tipo chamado de *TOperacao* e que foi definido como um conjunto de valores, esta situação é denominada como um *Enumerated Types* (tipos enumerados). Na linha 06 definimos o nome da nossa classe como *TMathClass*, e criamos quatro métodos de acordo com as operações matemáticas que queremos implementar (linhas 07 a 10), observe que definimos apenas o método *ExecutaOperacao* como parte da sessão *public*, dessa forma apenas este estará acessível para as demais classes do nosso projeto.

Após termos definido nossa classe, novamente pressione *Ctrl + Shift + C* para que os métodos sejam criados instantaneamente, começaremos criando o código da função de *Soma* conforme mostrado a seguir:

```
Result := Valor1 + Valor2;
```

Basicamente todas as rotinas serão idênticas, modificando apenas os sinais das operações matemáticas relativas, como mostra a implementação da rotina de *divisão*:

```
Result := Valor1 / Valor2;
```

A seguir a codificação da função de *Subtração:*

```
Result := Valor1 - Valor2;
```

Finalizando as rotinas de operação, criaremos a de multiplicação conforme a listagem seguinte:

```
Result := Valor1 * Valor2;
```

Deste modo, concluímos as rotinas de cálculos e agora faremos a implementação do método que realizará a chamada conforme a operação escolhida, perceba na listagem a seguir que podemos utilizar um comando de seleção para os tipos enumerados da mesma forma com que fazemos com inteiros por exemplo:

```
case Operacao of
  toSoma :          Result := Soma(Valor1, Valor2) ;
  toDivisao :       Result := Divisao(Valor1, Valor2);
  toSubtracao :     Result := Subtracao(Valor1, Valor2);
  toMultiplicacao : Result := Multiplicacao(Valor1, Valor2);
end;
```

Assim terminamos a nossa classe, para finalizar nosso artigo, faremos a última modificação do projeto, tornando possível o acesso do nosso formulário a esta classe criada. Sendo assim, adicione ao seu *Form* um *RadioGroup (rgOperacao),* adicionando à sua propriedade *Items* os valores *Soma, Subtração, Divisão, Multiplicação,* defina seu *Caption* como *Operações,* modifique para 0 à propriedade *ItemIndex* para que seja definido inicialmente a soma como operação padrão e por último altere a forma de exibição das colunas através da propriedade *Columns* e atribuindo o valor 2 à elas. Adicione agora três *Edit's (edtOp1, edtOp2, edtOpResultado)* e por último um *Button (btnCalcular).* Ao concluir, sua tela deverá ter uma aparência semelhante ao da **Figura 5.**

Adicione à cláusula *uses* do seu *Form* a *untFunctions,* para que possamos acessar a nossa classe criada. A última codificação do nosso projeto refere-se ao evento *OnClick* do *btnCalcular,* dessa forma adicione o código que se encontra na **Listagem 2**.

Perceba que na linha 02 temos a declaração de uma variável denominada como *Operacao* na qual referencia o tipo enumerado criado em nossa classe *TMathClass,* deste modo, podemos atribuir à variável operação as constantes anteriormente definidas. Entre as linhas 04 e 09 temos a escolha da *Operacao* de acordo com a posição da propriedade *ItemIndex* do nosso *RadioGroup,* dessa forma será tomada a decisão de escolha conforme a operação matemática selecionada. Por fim, da linha 10 em diante, temos a obtenção do resultado através do método *ExecutaOperacao* que é atribuído ao componente *edtOpResultado.* Salve seu projeto e execute o exemplo para visualizar os efeitos de nossa implementação.

Nota do DevMan

No desenvolvimento de softwares, um tipo enumerado é um tipo de dado que consiste em um conjunto de constantes pré-definidas e denominadas como enumeradores. Quando um identificador qualquer como uma variável, por exemplo, é declarada como um tipo enumerado pode-se atribuir um dos valores que foram definidos nas constantes a esta variável.

Nota

Caso queira que os métodos matemáticos estejam visíveis a qualquer parte do projeto, basta modificá-los, fazendo com que façam parte da sessão public ao invés da sessão private da classe TMathClass.

Figura 5. Aparência final do projeto.

Conclusão

Com isso finalizamos nosso artigo, como vimos é de grande importância e serventia o bom gerenciamento e entendimento dos métodos, para que assim consigamos desenvolver soluções mais organizadas, com maior eficácia e qualidade, tornando nossas tarefas cada vez mais práticas, produtivas e é claro garantindo maior facilidade na evolução do software em relação a futuras manutenções. Espero que tenham gostado. Um forte abraço e até o próximo artigo. ●

Dê seu feedback sobre esta edição!

A Clubedelphi tem que ser feita ao seu gosto. Para isso, precisamos saber o que você, leitor, acha da revista!

Dê seu voto sobre este artigo, através do link:

www.devmedia.com.br/clubedelphi/feedback

Conhecimento faz diferença!

Mais de 60 mil downloads na primeira edição!

Faça já sua assinatura digital! | *www.devmedia.com.br/es*

Faça um *up grade* em sua carreira.

Em um mercado cada vez mais focado em qualidade ter conhecimentos aprofundados sobre requisitos, metodologia, análises, testes, entre outros, pode ser a diferença entre conquistar ou não uma boa posição profissional. Sabendo disso a DevMedia lança para os desenvolvedores brasileiros sua primeira revista digital totalmente especializada em Engenharia de Software. Todos os meses você irá encontrar artigos sobre Metodologias Ageis; Metodologias tradicionais (*document driven*); ALM (*application lifecycle management*); SOA (aplicações orientadas a servicos); Analise de sistemas; modelagem; Métricas; orientação à objetos; UML; testes e muito mais. **Assine já!**

DevMedia GROUP

Nesta seção você encontra artigos sobre a linguagem PHP e a ferramenta Delphi for PHP

PHP com WebServices

Como criar e acessar WebServices utilizando o PHP?

Resumo do DevMan

Fazer a integração de dados entre empresas e ainda aumentar a segurança de um site e/ou aplicação Win32 não é uma tarefa muito simples, mas com o desenvolvimento de WebServices com certeza a tarefa torna-se menos trabalhosa.

WebServices são serviços que podemos disponibilizar na internet e que podem executar diversos tipos de tarefas, com a emissão de relatórios e ações de input e output. Eles podem ser desenvolvidos em praticamente qualquer linguagem. Nesse artigo veremos como trabalhar com ele usando PHP e duas extensões que nos possibilitam tal desenvolvimento.

Nesse artigo veremos
- Desenvolvimento de WebServices em PHP;
- Criação da aplicação Servidora;
- Criação da aplicação Cliente.

Qual a finalidade
Demonstrar o uso das extensões SOAP Extension e NuSOAP;

Quais situações utilizam esse recurso?
A criação e publicação de WebServices tem sido utilizada há tempos para fazer a integração de dados entre empresas e órgãos públicos além de aumentar a segurança de aplicações on-line

Adriano Santos
falecom@adrianosantos.pro.br
É desenvolvedor Delphi desde 1998, especialista em aplicações Win32, professor e programador PHP, bacharel em Comunicação Social pela Universidade Cruzeiro do Sul, SP. É Editor Chefe da revista ClubeDelphi e Editor Geral .net da WebMobile. Atua como Gerente de TI na SoftPark Informática e membro fundador do DUG-SP (www.dug-sp.com). Mantenedor do blog Delphi to Delphi (www.delphitodelphi.blogspot.com) com dicas, informações e tudo sobre o desenvolvimento Delphi.

A moda agora são os *WebServices*, pelo menos é o que parece. Realmente fazer uso dessa tecnologia nos ajuda muito no dia-a-dia. Já que é possível interligar diversos programas heterogêneos de maneira bem simples e organizada. O dia-a-dia do programador hoje é colocar o máximo de funcionalidades do aplicativo disponíveis para o mundo externo de maneira estruturada e simples. Isso porque é cada vez mais freqüente a troca de informações entre empresas, órgãos federais e governamentais, etc. A palavra *WebServices* tem sido citada constantemente, pois é a tecnologia que mais permite interligação entre várias plataformas.

Existem diversos WS's espalhados pela rede provendo os mais variados tipos de conteúdos. Além disso uma

série de vantagens são acrescidas ao software, como segurança por exemplo.

Nesse artigo veremos como desenvolver uma aplicação *WebService* usando o *PHP* puro. Como acessá-la e quais as vantagens disso tudo. Mão na massa.

Entendendo o negócio

Um WebService nada mais é do que uma DLL (*Dinamic Library Link*) e pode ser instalada no servidor/serviços de hospedagem. Nela incluímos funções de leitura e escrita. O mais comum é disponibilizarmos rotinas de leitura do banco de dados, e não de escrita. Mas é perfeitamente possível trabalhar com ambos na mesma aplicação. O mais interessante é que podemos manter uma boa comunicação entre diversos sistemas em diversas plataformas. Veja na **Figura 1** um diagrama mostrando a conexão entre dois servidores distintos.

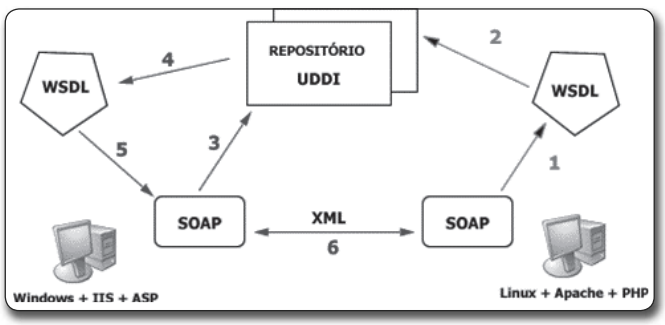


Figura 1. Diagrama de WebServices

Como vemos, temos o poder de ligar funções em servidores separados e plataformas completamente diferentes. É possível construir *WebServices* em praticamente todas as linguagens de programação, o que é ainda mais interessante, pois não precisamos nos matar de estudar uma linguagem apenas para desenvolver um pequeno *WebService*.

A primeira providência para gerar um *WebService*, também chamado de *WSDL*, é gerar um arquivo com as especificações desse módulo. Nele devem conter os nomes das funções, métodos, variáveis de entrada, saída, etc. Em seguida são gravados na *UDDI*.

Em seguida, você acessa a *UDDI* para pesquisar informações sobre determinados serviços. O *WSDL* tem acesso as funções de entrada e saída do seu *WebServices*. Feita a comunicação, entra em ação uma comunicação com protocolo *SOAP* em forma de *XML*.

Maneiras de acesso

Há basicamente duas formas de acessar *WebServices* com *PHP*. A partir da versão 5.0 do *PHP*, podemos utilizar uma a extensão *SOAP Extension*. Porém as limitações dessa extensão fazem com que seja necessário utilizarmos de outros métodos para acesso, como a biblioteca *NuSOAP*, a segunda opção. O *NuSOAP* na verdade é uma *API* desenvolvida especialmente para a linguagem. A principal vantagem dessa *API* é seu suporte embutido ao *WSDL*, *WebServices*.

Nota do DevMan

UDDI (originado do acrônimo inglês Universal Description, Discovery and Integration) é um protocolo aprovado como padrão pela OASIS e especifica um método para publicar e descobrir diretórios de serviços em uma arquitetura orientada a serviços (SOA).

Um serviço de registro UDDI é um Web Service que gerencia informação sobre provedores, implementações e metadados de serviços. Provedores de serviços podem utilizar UDDI para publicar os serviços que eles oferecem. Usuários de serviços podem usar UDDI para descobrir serviços que lhes interessem e obter os metadados necessários para utilizá-los.

A especificação UDDI define:

* API's SOAP utilizadas para publicar e obter informações de um registro UDDI;
* Esquemas XML do modelo de dados do registro e do formato das mensagens SOAP;
* Definições WSDL das API's SOAP;
* Definições de registro UDDI (modelos técnicos - TModels) de diversos sistemas de identificação e categorização, que podem ser utilizados para identificar e categorizar registros UDDI

Ambas maneiras são gratuitas e podem ser encontradas na internet para download. A primeira solução, SOAP Extension, pode ser baixada no endereço *http://phpsoaptoolkit.sourceforge.net/phpsoap/*. A segunda opção, NuSOAP, está disponível no endereço *http://sourceforge.net/projects/nusoap/*. A instalação e configuração de ambas soluções nós veremos nos tópicos dedicados a cada uma delas.

Utilizando SOAP Extension

Para entendermos ambas soluções disponíveis no mercado, faremos dois exemplos distintos. Cada um usando uma das soluções. Iniciaremos nosso primeiro exemplo usando SOAP Extension. A primeira coisa que precisamos fazer é baixar as extensão e instalar em nosso servidor. Acesse então o site da extensão mencionada através do link *http://phpsoaptoolkit.sourceforge.net/phpsoap/*, localize o arquivo necessário para instalação e baixe-o.

Após baixar o módulo copie-o para o diretório de instalação do PHP. O arquivo necessário tem o nome de *php_soap.dll* e precisa ser liberado no arquivo de configuração da linguagem, *php.ini*. Localize-o no diretório de instalação normalmente encontrado em *c:\arquivos de programas\php\php.ini*. Basta abrir a configuração e localizar a linha:

```
;extension=php_soap.dll
```

Para habilitar o acesso a extensão, apenas retire o ponto-e-vírgula da frente da linha. Caso ela não exista, apenas adicione a configuração e salve o arquivo.

Nota

Quando for necessário fazer a distribuição de sua aplicação utilizando as soluções apresentadas, entre em contato com o seu serviço de hospedagem para verificar se é necessário fazer tal configuração. Normalmente os serviços de hospedagem mantém arquivos de configuração (php.ini) separados para cada cliente.

Nota

A configuração do PHP 5.0 com Microsoft IIS (Internet Information Services) é uma tarefa não muito simples, por isso recomendo o uso de programas que auxiliares que criam o ambiente ideal para testes localmente. Um deles, já mencionado, é o Xampp. Ele pode se encontrado no link *http://xampp.sourceforge.net* é de fácil instalação e configuração.

Faremos a criação de um servidor bem simples. Apenas criaremos uma função que receberá um texto e fará a impressão no *Browser*. Por fim veremos a chamada ao servidor via aplicação *Cliente*.

Para escrever nossa aplicação, podemos usar um editor comum de textos como o *Bloco de Notas*, por exemplo ou programas específicos. É recomendado o *Adobe Dreamweaver* que inclusive possui função *highlight* que destaca as palavras reservadas da linguagem.

Inicie escrevendo o código da **Listagem 1** e vejamos como que funciona nossa aplicação. A primeira providência que fazemos em nossa aplicação servidora é criar uma instância do serviço para que possamos acessá-lo. Fazemos isso na **Linha 3** usando o método *new*. Repare que passamos para a classe a *url* de onde nosso servidor *SOAP* estará instalado. Em seguida criamos uma função chamada *imprimirtexto()* (**Linha 7**) que recebe como parâmetro o texto a ser impresso, como o próprio nome indica. Veja que o retorno da função é a frase *O texto enviado ao WebServices foi:
* concatenado ao parâmetro recebido. Incluímos também uma *tag HTML
* no texto para que tenhamos uma quebra de linha entre a frase e o texto recebido. Salve o arquivo como *servidor.php* no diretório de nosso *Web Site*.

Na **Linha 12** fazermos o registro da função no servidor. Veja que enviamos o nome da função para o método *addFunction* entre *strings*, é uma característica comum no *PHP*. Por fim, entre as **Linhas 17** e **22**, fazemos a chamada ao método servidor para atender a requisição feita. Repare que verificamos se o método utilizado é um POST, então chamamos o servidor. Caso contrário, apenas imprimimos as funções disponíveis no *WebServices*. Isso é importante, pois dessa forma o desenvolvedor que conhecer nosso *WebServices* por fazer uma pequena chamada ao servidor para descobrir quais funções e parâmetros estão disponíveis.

Chamando o servidor pela aplicação cliente

Nosso servidor está pronto, agora precisamos ver como fazer para chamar as funções nele contidas. Isso é feito pela nossa aplicação cliente, ou seja, pela a aplicação que será visualizada pelo usuário final, o internauta. Veremos que é muito simples fazer a chamada. Então, crie um novo arquivo no seu editor de textos ou de código preferido e escreva o código da **Listagem 2** e salve-o como *cliente.php*.

Muito bem, vamos entender o que fazemos em nossa aplicação cliente. A providência número 1 é criar uma variável associando a ela uma instância do servidor. Para isso usamos os métodos *new* e *SoapClient*. Perceba que passamos como parâmetro a localização do arquivo que acabamos de desenvolver, *servidor.php*. Passamos também a uri do servidor *SOAP* que na verdade é apenas a raiz do nosso servidor.

Na **Linha 9** fazemos a chamada definitiva da função passando para ela um texto. No caso passei meu próprio nome. Essa é sem dúvida a parte mais interessante de nosso sistema. Observe que apenas passamos para a função os parâmetros necessários para que o *WebServices* execute a ação desejada. Em uma aplicação real, é nesse momento que ganhamos mais inteligência no sistema. Por fim fazemos um pequeno tratamento de erros. Caso um erro seja encontrado, imprimimos a *string* em tela.

Claro que esse é um exemplo bem simples e no lugar de nossa função *imprimirtexto()* podemos fazer *n* operações. Como funções de *input/output* no banco de dados, cálculos diversos e todo tipo de procedimento que nossas aplicações necessitarem.

Para testar o *Web Site*, apenas acesse o endereço do arquivo *cliente.php*.

Usando o módulo NuSOAP

Acabamos de ver como criar um *WebServices* em *PHP* e acessá-lo usando a extensão *SOAP Extension*. A utilização dela é bastante simples como pudemos notar. Agora, vejamos como fazer a mesma aplicação usando a segunda solução proposta, *NuSOAP*.

Listagem 1. Código do servidor da aplicação

```
01 <?php
02 /* Criamos primeiramente a instância para acesso ao servidor */
03 $servidor = new SoapServer(null, array('uri' =>
04 "http://localhost/clubedelphi/soap"));
05
06 /* Definimos a função do serviço */
07 function imprimirtexto($texto){
08      return 'O texto enviado ao WebServices foi:<br> ' . $texto;
09 }
10
11 /* Registramos o serviço */
12 $server->addFunction("imprimirtexto");
13 /* Aqui chamamos o método do servidor para atender a
     requisição do serviço. Quando
14   uma chamada via POST é feita, então a executamos, caso
      contrário apenas mostramos 15 os métodos cadastrados*/
16
17   if ($_SERVER["REQUEST_METHOD"]== "POST"){
18        $server->handle();
19   }else{
20        $functions = $server->getFunctions();
21        foreach($functions as $func){
22           print $func . "<br>";
23   }
24 }
25 ?>
```

Listagem2. Aplicação cliente

```
01 <?php
02 /* Criamos primeiramente a instância para acesso ao servidor */
03 $cliente = new SoapClient(null, array(
04      'location' => 'http://localhost/clubedelphi/soap/
          servidor.php',
05      'uri' => 'http://localhost/clubedelphi/soap',
06      'trace' => 1)
07 );
08 /* Chamamos o serviço SOAP client */
09 $result = $client->imprimirtexto("Adriano Santos");
10
11 /* Verificar erros de execução do serviço */
12 if (is_soap_fault($result)){
13      trigger_error("SOAP Fault: (faultcode: {$result->
          faultcode},
14          faultstring: {$result->faultstring})", E_ERROR);
15 }else{
16      echo "Resultado Encontrado: <br><br>";
17          print_r($result);
18 }
19 ?>
```

Temo que iniciar fazer o download e instalação. Na verdade a utilização dessa extensão é ainda mais simples do que a que acabamos de ver. Basta fazer um *include()* ou *require_once()* do arquivo *nusoap.php* em nosso servidor.

Nota

Como iremos utilizar uma outra extensão para desenvolver nosso WebServices, é necessário desabilitar a extensão SOAP Extension. Para fazer isso é muito simples. Basta abrir novamente o arquivo php.ini e comentar (Incluir um ponto-e-vírgula na frente da frase) ou excluir a linha extension=php_soap.dll.

Agora precisamos desenvolver um novo servidor, portanto crie uma nova pasta para nossa aplicação e crie um novo arquivo chamado *servidor.php*. O seu código pode ser visto da **Listagem 3** e é bem fácil de entendermos.

Repare que fazemos uma chamada ao método *include_once()*, que adiciona o arquivo *nusoap.php* ao código do servidor. Na verdade a extensão *NuSOAP* é uma classe, por isso adicionamos diretamente seu arquivo.

Em seguida precisamos instanciar nossa classe usando o método *new*, assim como já vimos em outros artigos e aqui mesmo nas listagens anteriores. Registramos o serviço na **Linha 9** e confirmamos a requisição nas demais linhas. Como podemos ver não há segredos.

Agora crie um novo arquivo e digite o código da **Listagem 4** que contém a aplicação cliente.

A aplicação cliente é ainda mais simples. Novamente incluímos a classe *NuSOAP* e em seguida instanciamos o cliente usando o método *new*. Chamamos o serviço usando o método *call* passando como parâmetro o texto a ser impresso. Por fim imprimimos o resultado usando *print_r()*.

Se executarmos a aplicação nesse momento, teremos o mesmo resultado do exemplo anterior provando que ambas soluções são válidas e muito simples de serem implementadas.

Listagem 3. Código da aplicação servidora com NuSOAP

```
01 <?php
02 /* Inclui a classe NuSOAP */
03 require_once('nusoap.php');
04
05 /* Criamos uma instância do servidor */
06 $servidor = new soap_server;
07
08 /* Registramos o(s) método(s) */
09 $server->register('imprimirtexto');
10
11 /* Definimos o método */
12      function imprimirtexto($texto){
13         return 'O texto enviado ao WebServices foi:<br> ' . $texto;
14      }
15
16 /* Registrando a requisição */
17 $HTTP_RAW_POST_DATA = isset($HTTP_RAW_POST_DATA)
18 $HTTP_RAW_POST_DATA : '';
19 $servidor->service($HTTP_RAW_POST_DATA);
20 ?>
```

Listagem 4. Código da aplicação cliente com NuSOAP

```
01 <?php
02 /* Inclusão do arquivo de Classes NuSOAP */
03 require_once('nusoap.php');
04
05 /* Instância da classe cliente */
05 $cliente = new soapclient('http://localhost/clubedelphi/
     nuSOAP/servido.php');
06
07 /* Chamada ao método propriamente dito */
08 $resultado = $cliente->call('imprimirtexto', array('Adriano Santos'));
09
10 /* Imprimimos o resultado */
11 print_r($resultado);
12 ?>
```

Considerações finais

Qual a real necessidade em se criar *WebServices*? Para quem não sabe exatamente qual é a real utilidade de um *WebServices*, vamos esclarecer algumas dúvidas. Um *WebServices*, como o próprio nome sugere, é um serviço *Web*, ou seja, podemos criar todo tipo de serviço na internet para atender as mais diversas necessidades.

Os *Correios*, por exemplo, disponibilizam *WebServices* para consulta de *CEP*, consulta de encomendas, e uma série de outras coisas. Dessa forma, os *Correios* não precisam criar *n* formas para expor seus dados, bastando criar um *WebServices* e publicar a documentação dele. Por sua vez, cada desenvolvedor precisa apenas conhecer os parâmetros necessários para usufruir de suas funções.

Além disso, o nível de segurança é estendido, já que é mais difícil burlar um site que possui suas principais funções publicadas em *WebServices*. Enfim, um *WebServices* pode ser criado em qualquer linguagem de programação, até em uma linguagem de script como o *PHP*.

Conclusão

Como pudemos notar, desenvolver uma aplicação capaz de trabalhar em conjunto com *WebServices* desenvolvido em *PHP*, não é nenhum bicho de sete cabeças e portanto muito simples de se desenvolvido.

Vimos que há basicamente duas formas de desenvolver um *WS* com *PHP*. Eu sugiro que o leitor procure estudar mais essa tecnologia que tem crescido bastante nos últimos tempos, inclusive para fazer a integração entre sistemas *Desktop*, *Win32*. Na internet há muito material disponível para estudar essa tecnologia.

Um forte abraço e até a próxima. ●

Dê seu feedback sobre esta edição!

A Clubedelphi tem que ser feita ao seu gosto. Para isso, precisamos saber o que você, leitor, acha da revista!

Dê seu voto sobre este artigo, através do link:

www.devmedia.com.br/clubedelphi/feedback

Trabalhando com transações avançadas em DBExpress e DataSnap

Aprenda a reaproveitar as transações já abertas pelo DataSnap

Nesse artigo veremos

• Modelagem e criação de banco de dados.

• Utilização de recursos da tecnologia DataSnap para controle transacional;

• Técnicas que podem agregar qualidade e facilidade na manutenção do software.

Qual a finalidade

Utilizar os recursos transacionais da tecnologia DataSnap evitando os controles transacionais manuais, aumentando a produtividade e facilidade de uso dos mesmos.

Quais situações utilizam esse recurso?

Qualquer aplicação de arquitetura cliente-servidor ou multicamadas que utilize a tecnologia DataSnap, podendo ser aplicável a qualquer tipo de software que necessite realizar ações seqüenciais como atualizar estoque ou gerar contas a pagar / receber por exemplo.

Daniel Sobrinho Laporte

daniel.laporte@gmail.com

É graduado em sistemas de informação e pós-graduado em administração de bancos de dados pela Universidade de Franca. É analista de sistemas e desenvolvedor Delphi desde 2005 desenvolvendo aplicações cliente/servidor e multicamadas em Delphi com Firebird.

Resumo do DevMan

Qualquer desenvolvedor iniciante ou não já teve dúvidas de como utilizar uma transação para realizar ações após o término de uma ação prévia. Transações requerem certos cuidados, pois podem ocasionar sérios problemas a um software caso não sejam utilizadas corretamente. Neste artigo é apresentada uma forma elegante e diferente de realizar atualizações / inserções por meio de uma transação aberta pelo próprio Datasnap, prevenindo problemas com transações manuais e garantindo maior facilidade para a manutenção.

Por participar muito de listas de discussão, percebo que alguns assuntos sempre são pauta nos fóruns e abordam questões do tipo: *Como gerar meu contas a receber após a venda usando multicamadas?* ou *Como utilizo minha transação para atualizar meu estoque ao concluir/cancelar a venda?*.

Grande parte dos desenvolvedores fazem estas tarefas utilizando um conjunto de objetos de bancos de dados como *Triggers* e *Procedures*, outros criam métodos na própria linguagem, passando parâmetros e montando a instrução de inserção ou atualização manualmente.

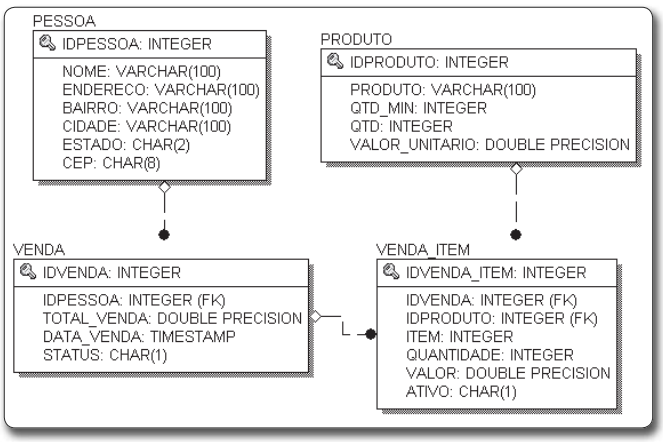


Figura 1. Modelagem proposta para o nosso exemplo

Neste artigo irei abordar uma maneira diferente de se realizar tarefas como estas utilizando a própria transação do *DataSnap*, através de um exemplo comumente utilizado em sistemas comerciais, o típico controle de estoque no ato da venda.

Entendendo a problemática do sistema

Para o nosso exemplo faremos com que o usuário cadastre os produtos e as pessoas, *clientes*, que serão necessários para simularmos um exemplo simples de venda. O detalhe fica no cadastro de produtos onde informaremos a quantidade disponível de determinado produto, para que, no ato da venda façamos a baixa automática do estoque de cada produto vendido sem ter que iniciar uma transação manualmente. Ao efetuar o cancelamento da venda o estoque será automaticamente reposto com os produtos da venda cancelada.

Modelando e criando o banco de dados

Para o desenvolvimento de nosso banco utilizei o banco de dados *Firebird* em sua versão 1.5.5, porém o exemplo é totalmente aplicável a qualquer outra versão do mesmo. Sem mais delongas vamos à **Figura 1.**

Analisando-a, é possível ver que nossa modelagem é bem simples e composta de quatro tabelas sendo: *PESSOA, PRODUTO, VENDA e VENDA_ITEM*. É claro que algumas tabelas não foram implementadas por não serem o foco do nosso artigo, porém, através deste exemplo poderemos simular a atualização de estoque através. Vale lembrar que este exemplo pode ser aplicado a inúmeras situações e em diferentes arquiteturas que utilizem *DataSnap.*

A **Listagem 1** contém todo o script do banco de dados. Para a criação do seu banco de dados, basta utilizar uma ferramenta como o *IBExpert* ou *IBOConsole* e executar o mesmo no local desejado. Passaremos agora para o desenvolvimento da nossa aplicação.

Nota

Em caso de erro na atualização do estoque ou mesmo na própria venda / cancelamento, toda a operação será revertida do mesmo modo que o rollback de uma transação implementada manualmente, caso contrário, todas as operações receberão o commit realizado automaticamente pelo DataSnap.

Criando o modelo de aplicação

Enfim vamos dar inicio ao desenvolvimento da nossa aplicação. Abra seu *Rad Studio 2007* e acesse o menu *File>New>VCL Forms Application – Delphi for Win32* para criar um novo projeto, em seguida altere as propriedades *Name* para *frmMenu*, *Caption* para *Controle de Estoque*, *Position* para *poDesktopCenter*, *Height* para *600* e *Width* para *800* e adicione um componente *MainMenu* e um *ImageList*. Altere sua propriedade *Name* para *imlImagens* e dê dois clique sobre o mesmo para adicionar algumas imagens de sua preferência relacionando-os com produtos, pessoas e vendas. Em seguida, aponte o *imlImagens* à propriedade *Images* do componente *MainMenu*. Dê um duplo clique sobre o componente *MainMenu*, criando um novo menu através da tecla *Insert* e

Listagem 1. Script do banco de dados

```
01 SET SQL DIALECT 3;
02 SET NAMES WIN1252;
03 SET CLIENTLIB '<DIRETÓRIO DA INSTALAÇÃO DO FIREBIRD>\BIN\
    fbclient.dll';
04 CREATE DATABASE '<DIRETÓRIO DO BANCO DE DADOS>\VENDAS.FDB'
05 USER 'SYSDBA' PASSWORD 'masterkey'
06 PAGE_SIZE 4096
07 DEFAULT CHARACTER SET WIN1252;

08 CREATE TABLE PESSOA
09 ( IDPESSOA INTEGER NOT NULL,
10   NOME VARCHAR(100) NOT NULL,
11   ENDERECO VARCHAR(100),
12   BAIRRO VARCHAR(100),
13   CIDADE VARCHAR(100),
14   CEP CHAR(8),
15   ESTADO CHAR(2));

16 ALTER TABLE PESSOA ADD CONSTRAINT PK_PESSOA
17 PRIMARY KEY (IDPESSOA) USING INDEX IDX_PK_PESSOA;

18 CREATE TABLE PRODUTO
19 ( IDPRODUTO INTEGER NOT NULL,
20   PRODUTO VARCHAR(100) NOT NULL,
21   QTD_MIN INTEGER NOT NULL,
22   QTD INTEGER NOT NULL,
23   VALOR_UNITARIO DOUBLE PRECISION NOT NULL);

24 ALTER TABLE PRODUTO ADD CONSTRAINT PK_PRODUTO
25 PRIMARY KEY (IDPRODUTO) USING INDEX IDX_PK_PRODUTO;

26 CREATE TABLE VENDA
27 ( IDVENDA INTEGER NOT NULL,
28   IDPESSOA INTEGER NOT NULL,
29   TOTAL_VENDA DOUBLE PRECISION NOT NULL);

30 ALTER TABLE VENDA ADD CONSTRAINT PK_VENDA
31 PRIMARY KEY (IDVENDA) USING INDEX IDX_PK_VENDA;

31 ALTER TABLE VENDA ADD CONSTRAINT FK_VENDA_PESSOA
32 FOREIGN KEY (IDPESSOA) REFERENCES PESSOA(IDPESSOA)
33 USING INDEX IDX_VENDA_PESSOA;

34 CREATE TABLE VENDA_ITEM
35 ( IDVENDA_ITEM INTEGER NOT NULL,
36   IDVENDA INTEGER NOT NULL,
37   IDPRODUTO INTEGER NOT NULL,
38   ITEM NOT NULL,
39   VALOR DOUBLE PRECISION NOT NULL,
40   QUANTIDADE INTEGER NOT NULL);

41 ALTER TABLE VENDA_ITEM ADD CONSTRAINT PK_VENDA_ITEM
42 PRIMARY KEY (IDVENDA_ITEM) USING INDEX IDX_VENDA_ITEM;

43 ALTER TABLE VENDA_ITEM ADD CONSTRAINT FK_ITEM_VENDA
44 FOREIGN KEY (IDVENDA) REFERENCES VENDA(IDVENDA)
45 ON DELETE CASCADE
46 ON UPDATE CASCADE
47 USING INDEX IDX_ITEM_VENDA;

48 ALTER TABLE VENDA_ITEM ADD CONSTRAINT FK_VENDA_PRODUTO
49 FOREIGN KEY (IDPRODUTO) REFERENCES PRODUTO(IDPRODUTO)
50 USING INDEX IDX_VENDA_PRODUTO;

51 CREATE GENERATOR GEN_IDPRODUTO;
52 CREATE GENERATOR GEN_IDPESSOA;
53 CREATE GENERATOR GEN_IDVENDA;
54 CREATE GENERATOR GEN_IDVENDA_ITEM;
```

altere sua propriedade *Caption* para *Cadastro,* adicione dentro do mesmo outros dois itens de menu relativos a *Produtos* e *Pessoas*, repita o processo criando um menu chamado *Financeiro* e um sub-item *Vendas* e por último o menu *Outros* com sub-item *Sair*, para exibir os *glyphs* nos menus criados simplesmente altere suas propriedades *ImageIndex* indicando o item correspondente. Após concluir estas alterações acesse o menu *File>Save All* nomeando a Unit como *untMenu* e o projeto como *Estoque.dpr*. Veja o resultado final do nosso formulário principal na **Figura 2**.

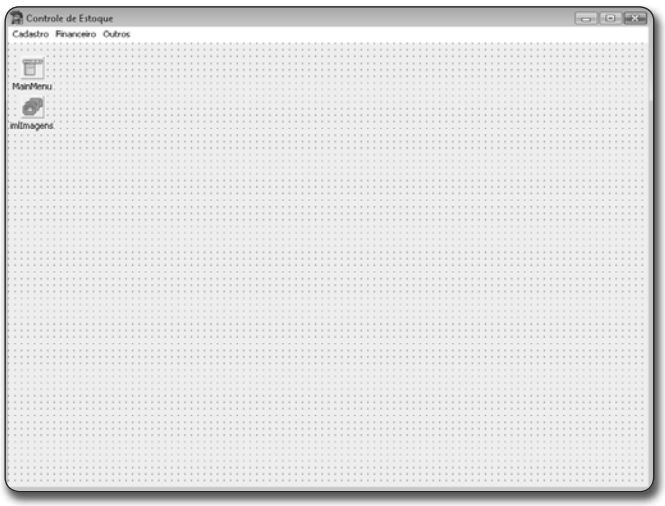


Figura 2. Formulário principal da aplicação

Iniciaremos pelo desenvolvimento do *Data Module*¸ pois este será fundamental para a criação dos nossos formulários. No entanto apenas adicionaremos os componentes e faremos as implementações mais simples e necessárias para seguirmos adiante, deixando a parte mais interessante para o final do artigo onde iremos implementar a venda.

Acesse o menu *File>New>Other>Delphi Files* e do lado direito selecione *Data Module.* Altere sua propriedade *Name* para *DMPrincipal* e adicione à sua cláusula *uses* a Unit *Variants*. Adicione ao novo *Data Module* um *SQLConnection,* seis *SQLDataSet,* cinco *ClientDataSet,* quatro *DataSetProvider* e por último um *DataSource*. Vamos iniciar as configurações pelo *SQLConnection*. Selecione o componente e altere suas propriedades *Name* para *SQLConexao* e *LoginPrompt* para *False* e por último *VendorLib* para *fbclient.dll*, em seguida dê um duplo clique no mesmo para abrir a janela de configuração de conexão e configure-a conforme mostra a **Figura 3.**

Feito isso, iremos configurar os *SQLDataSet*. Inicie alterando suas propriedades *Name* para *sqlPessoa*, *sqpProduto*, *sqlVenda*, *sqlVenda_Item*, *sqlPesquisa* e *sqlAtualizacao* respectivamente. Selecione todos e aponte suas propriedades *SQLConnection* ao *SQLConexao*, em seguida, adicione as sentenças *SQL* através da propriedade *CommandText* conforme mostrado na **Listagem 2.**

ClubeDelphi PLUS www.devmedia.com.br/clubedelphi/portal.asp

Acesse agora mesmo o portal do assinante ClubeDelphi e veja uma vídeo aula de Luciano Pimenta que mostra como utilizar o IBExpert para administração avançada de banco de dados Firebird/Interbase.

http://www.devmedia.com.br/articles/viewcomp.asp?comp=3082

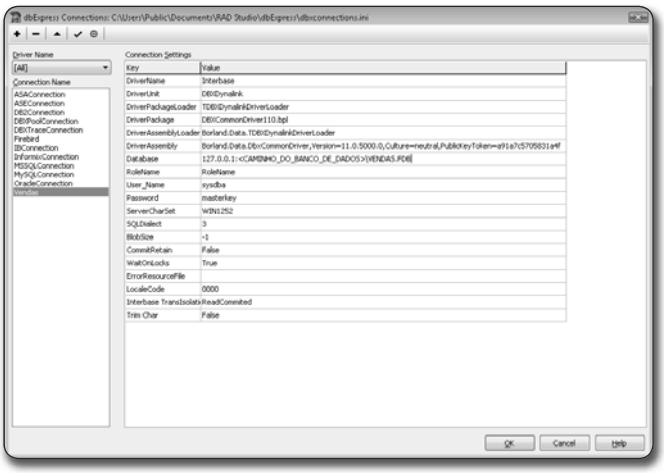


Figura 3. Exemplo de configuração do banco de dados no dbexpress

Listagem 2. Sentenças SQL dos componentes SQLDataSet

```
[sqlPessoa]
SELECT
  PESSOA.IDPESSOA,
  PESSOA.NOME,
  PESSOA.ENDERECO,
  PESSOA.BAIRRO,
  PESSOA.CIDADE,
  PESSOA.ESTADO,
  PESSOA.CEP
FROM
  PESSOA
WHERE
  PESSOA.IDPESSOA =:IDPESSOA

[sqlProduto]
SELECT
  PRODUTO.IDPRODUTO,
  PRODUTO.PRODUTO,
  PRODUTO.QTD_MIN,
  PRODUTO.QTD,
  PRODUTO.VALOR_UNITARIO
FROM
  PRODUTO
WHERE
  PRODUTO.IDPRODUTO =:IDPRODUTO

[sqlVenda]
SELECT
  VENDA.IDVENDA,
  VENDA.IDPESSOA,
  VENDA.TOTAL_VENDA,
  VENDA.DATA_VENDA,
  VENDA.STATUS,
  PESSOA.NOME
FROM
  VENDA
LEFT JOIN PESSOA ON
  VENDA.IDPESSOA = PESSOA.IDPESSOA
WHERE
  VENDA.IDVENDA =:IDVENDA

[sqlVenda_Item]
SELECT
  VENDA_ITEM.IDVENDA_ITEM,
  VENDA_ITEM.IDVENDA,
  VENDA_ITEM.IDPRODUTO,
  VENDA_ITEM.ITEM,
  VENDA_ITEM.VALOR,
  VENDA_ITEM.QUANTIDADE,
  VENDA_ITEM.ATIVO,
  PRODUTO.PRODUTO,
  PRODUTO.VALOR_UNITARIO,
  PRODUTO.QTD
FROM
  VENDA_ITEM
LEFT JOIN PRODUTO ON
  VENDA_ITEM.IDPRODUTO = PRODUTO.IDPRODUTO
WHERE
  VENDA_ITEM.IDVENDA=:IDVENDA
```

Após ter adicionado as sentenças *SQL* aos respectivos componentes, dê um clique na propriedade *Params* para que o editor de parâmetros seja exibido. Deste modo, selecione o parâmetro disponível e do lado esquerdo no *Object Inspector* configure o *DataType* para *ftInteger* em todos os componentes. A última etapa a fazer é selecionar os *SQLDataSet* dando um duplo clique do botão esquerdo do *mouse* para fazer com que o *Fields Editor* seja apresentado. Adicione os campos do *sqlPessoa* e *sqlProduto*, e campo a campo faça as devidas configurações de *ProviderFlags*, marcando os campos chave primária com as opções *[pfInKey, pfInUpdate, pfinWhere]*, os campos que deverão ser atualizados devem ser marcados como *[pinUpdate]* e para os demais, desmarque todas as opções.

Antes de configurarmos os *ProviderFlags* dos componentes *sqlVenda* e *sqlVenda_Item*, devemos nos atentar ao *DataSource*, pois este será responsável por fazer a ligação entre os dois por meio de um *mestre-detalhe*, portanto, altere sua propriedade *Name* para *dsrVenda* e aponte sua propriedade *DataSet* ao *sqlVenda (mestre)*. Em seguida, no componente *sqlVenda_Item* configure a propriedade *DataSource* apontando para o *dsrVenda*. Deste modo, realizamos a ligação do *mestre-detalhe* e podemos prosseguir com a configuração dos *Fields* dos dois componentes restantes do mesmo modo em que fizemos nos anteriores.

Vamos passar agora para as configurações dos componentes *DataSetProvider*. Inicie alterando suas propriedades *Name* para *dspPessoa*, *dspProduto*, *dspVenda* e *dspPesquisa* e em *DataSet* ligue-os aos respectivos *sqlDataSet*. Todo o processo de persistência de dados é realizado conforme a definição da opção *UpdateMode*. Como desejamos que este processo seja o mais produtivo possível, marque em todos para que a maneira de atualizar seja *upWhereKeyOnly*. Por último, vamos configurar as propriedades *Options* de cada um. Selecione primeiramente os componentes *dspPessoa* e *dspProduto* e modifique as propriedades *Options* habilitando para *True* as opções *[poIncFieldProps, poPropagateChanges]*, para o componente *dspVenda* habilite *[poIncFieldProps, poPropagateChanges, poCascadeDeletes, poCascadeUpdates]* e por último no *dspPesquisa* marque apenas a opção *[poAllowCommandText]* finalizando assim esta parte.

Para os *ClientDataSet* altere as propriedades *Name* para *cdsPessoa*, *cdsProduto*, *cdsVenda*, *cdsVenda_Item* e *cdsPesquisa*, ligue primeiramente o *dspVenda* à propriedade *ProviderName* do componente *cdsVenda* e em seguida dê um duplo clique sobre o mesmo para abrir o *Fields Editor*. Clique sobre ele com o botão direito do *mouse* e selecione a opção *Add all Fields*. Se suas ligações estiverem corretas você perceberá um campo do tipo *DataSetField* chamado *sqlVenda_Item*. Ligue-o na propriedade *DataSetField* do *cdsVenda_Item* para que o relacionamento *mestre-detalhe* seja finalizado e em seguida adicione os *Fields* do *cdsVenda_Item* do mesmo modo que foi feito anteriormente. Nos *ClientDataSet* restantes faça as atribuições dos respectivos *DataSetProviders* através da propriedade *ProviderName* e adicione os *Fields* nos mesmos, desprezando este último processo de adição dos campos apenas para o *cdsPesquisa*.

Assim finalizamos as ligações de todos os componentes do *Data Module*. Agora que concluímos esta parte vamos codificar os métodos necessários para o seu funcionamento. Primeiramente, declare na sessão *public* o cabeçalho da função apresentada na **Listagem 3,** pressione as teclas *Ctrl + Shift + C* simultaneamente para que o *Delphi* crie o corpo do método automaticamente.

A rotina da **Listagem 3,** como o próprio nome sugere, retorna o valor do *Generator* definido no banco de dados. Entre as linhas 02 e 05 temos a declaração dos objetos necessários para realizar esta busca de valores diretamente do banco de dados, em seguida das linhas 09 a 12 temos a criação dos objetos dinamicamente e a conexão do componente

Listagem 3. Método GetGenerator

```
01  function TDMPrincipal.GetGenerator(ChavePrimaria: String):
     Integer;
02  var
03    cds : TClientDataSet;
04    dsp : TDataSetProvider;
05    sql : TSQLDataSet;
06  begin
07    try
08      try
09        cds := TClientDataSet.Create(Self);
10        dsp := TDataSetProvider.Create(Self);
11        sql := TSQLDataSet.Create(Self);
12        sql.SQLConnection := SQLConexao;
13        sql.CommandText :=
14          'SELECT GEN_ID(GEN_' + ChavePrimaria + ',1) FROM
              RDB$DATABASE';
15        dsp.Options := [poAllowCommandText];
16        dsp.DataSet := sql;
17        cds.SetProvider(dsp);
18        cds.Open;
19        Result := cds.Fields[0].AsInteger;
20      except
21        Result := -1;
22      end;
23    finally
24      FreeAndNil(cds);
25      FreeAndNil(dsp);
26      FreeAndNil(sql);
27    end;
28  end;
```

Nota do DevMan

A propriedade TDataSetProvider.UpdateMode é utilizada para gerar a sentença SQL que será utilizada na verificação da situação do registro em foco, isto é, se o mesmo foi alterado desde que foi recuperado do banco de dados. Temos os seguintes modos:

upWhereAll – Esta opção faz com que todos os campos que possuem a propriedade ProviderFlags em pfInWhere sejam adicionados na cláusula where que será gerada para localização do registro;

upWhereChanged – Esta faz com que sejam utilizados apenas os campos que foram alterados;

upWhereKeyOnly – Para localizar o registro será utilizada apenas sua chave primária.

Nota

Se você habilitou as opções poIncFieldProps e poPropagateChanges nos componentes DataSetProvider, perceba que as alterações relativas à máscaras, Display e configurações dos ProviderFlags por exemplo, são automaticamente propagadas ao ClienteDataSet simplesmente setando sua propriedade Active para True.

sql do tipo *SQLDataSet* com o banco de dados por meio do *SQLConnection* (*sqlConexao)*. Perceba que é realizada uma consulta utilizando a tabela *RDB$DATABASE* como base, juntamente com a concatenação do campo que é denominado chave primária, passado por parâmetro. A seguir é habilitada a opção *[poAllowCommandText]* no *DataSetProvider* que nos permite realizar uma consulta através do *ClientDataSet*, linha 15. O resultado da consulta é retornado na linha 19. Devido ao fato do nosso código estar isolado dentro de uma sessão de exceção, em caso de erro automaticamente o valor retornado será -1 conforme mostra a linha 21. E por último entre as linhas 24 a 26 temos a destruição e liberação dos componentes da memória.

Feito isso, partiremos agora para os demais métodos. Selecione o componente *cdsPessoa* e no evento *AfterInsert* adicione o código a seguir, responsável por atribuir o valor da chave primária correspondente.

```
procedure TDMPrincipal.cdsPessoaAfterInsert(DataSet: TDataSet);
begin
  cdsPessoaIDPESSOA.AsInteger := GetGenerator('IDPESSOA');
end;
```

Implemente o mesmo evento porém agora no componente *cdsProduto* conforme mostrado:

```
procedure TDMPrincipal.cdsProdutoAfterInsert(DataSet: TDataSet);
begin
  cdsProdutoIDPRODUTO.AsInteger := GetGenerator('IDPRODUTO');
end;
```

Deste modo finalizamos o desenvolvimento inicial do *DMPrincipal* da aplicação, lembrando que mais adiante ao implementarmos a venda, voltaremos para concluir as codificações restantes e consequentemente exibir suas particularidades. O exemplo do *Data Module* pronto pode ser visualizado na **Figura 4.** Terminada esta etapa, vá em *File>Save All* e dê o nome à *unit* de *untDMPrincipal.pas* para darmos inicio a criação dos formulários de cadastro e consulta.

Iremos desenvolver nossa aplicação utilizando a herança de formulários, pois, é um recurso que nos torna extremamente produtivos no desenvolvimento, além de facilitar muito a manutenção devido à centralização do código.

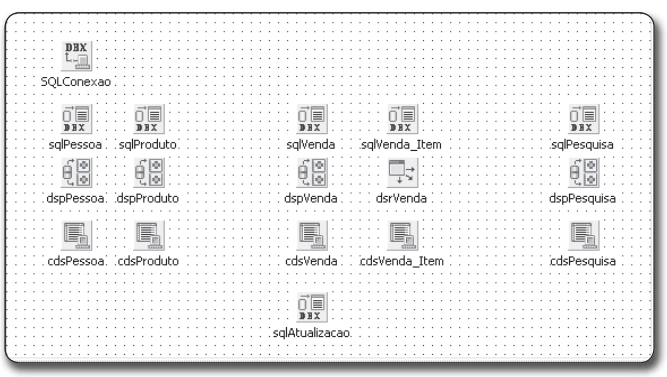


Figura 4. Data Module principal devidamente configurado

Em nossa aplicação vamos criar dois modelos de herança. O primeiro será responsável pelos cadastros e o segundo será responsável pelas consultas, e no decorrer da aplicação faremos a união dos dois ao criarmos os cadastros padrões.

Acesse o menu *File>New>Form – Delphi for Win32* para criar um novo formulário, altere suas propriedades *Name* para *frmCadastro, Position* para *poMainFormCenter, Height* para *332* e *Width* para *616.* Em seguida adicione um componente *Panel* alterando sua propriedade *Align* para *alTop,* um *DataSource,* alterando seu *Name* para *dsrCadastro* e sete componentes *BitBtn* alterando suas proprieades *Name* e *Caption* para *btnIncluir e Incluir, btnEditar e Editar, btnExcluir e Exlcuir, btnCancelar e Cancelar, btnSalvar e Salvar, btnPesquisar e Pesquisar* e *btnSair e Sair,* respectivamente. Para este último altere sua propriedade *ModalResult* para *mrOk*. Veja como ficou o *layout* do nosso formulário na **Figura 5.**

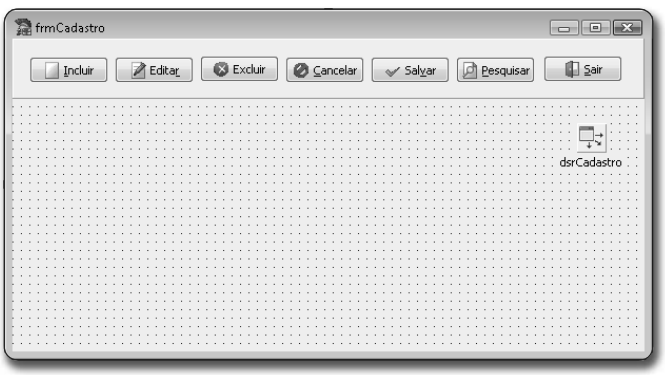


Figura 5. Modelo de formulário de herança para cadastros

Antes de implementar os eventos necessários, adicione à cláusula *Uses* do seu *Form* a Unit *DBClient*. Seguindo a ordem dos botões, iremos codificar os eventos *OnClick* de cada *BitBtn* de acordo com suas ações, portanto, comece adicionando no *btnIncluir* o código abaixo que simplesmente verifica se o *DataSet* vinculado ao *dsrCadastro* está ativo, e em seguida faz a inclusão*:*

```
procedure TfrmCadastro.btnIncluirClick(Sender: TObject);
begin
  if not dsrCadastro.DataSet.Active then
    dsrCadastro.DataSet.Open;
  dsrCadastro.DataSet.Append;
end;
```

O evento *OnClick* do botão editar é bem semelhante porém, verifica se o *DataSet* não se encontra vazio antes de editá-lo.

```
procedure TfrmCadastro.btnEditarClick(Sender: TObject);
begin
  if not dsrCadastro.DataSet.IsEmpty then
    dsrCadastro.DataSet.Edit;
end;
```

Nota

É importante toda vez que utilizar componentes criados em tempo de execução realizar a liberação dos mesmos da memória, caso contrário, podemos ter sérios problemas com vazamentos de memória.

Alguns eventos fazem uma pergunta de confirmação da ação ao usuário como o caso do botão excluir, que verifica se o usuário quer realmente realizar esta ação, sendo verdadeira a condição, o registro é excluído e já gravado no banco de dados como mostra a **Listagem 4**.

Listagem 4. Exclusão

```
01 procedure TfrmCadastro.btnExcluirClick(Sender: TObject);
02 begin
03   if not dsrCadastro.DataSet.IsEmpty then
04   begin
05      if Application.MessageBox('Deseja excluir ?', 'Confirmação !',
06        MB_YESNO + MB_ICONQUESTION) = ID_YES then
07        begin
08          dsrCadastro.DataSet.Delete;
09          if TClientDataSet(dsrCadastro.DataSet).
             ApplyUpdates(0) <> 0 then
10            Application.MessageBox('Problemas na exclusão !',
              'Erro !',
11              MB_ICONERROR + MB_OK);
12        end;
13    end;
14 end;
```

O mesmo ocorre no evento *onClick* do *btnCancelar* como é mostrado a seguir.

```
procedure TfrmCadastro.btnCancelarClick(Sender: TObject);
begin
  if Application.MessageBox('Deseja cancelar ?', 'Confirmação !',
    MB_YESNO + MB_ICONQUESTION) = ID_YES then
    dsrCadastro.DataSet.Cancel;
end;
```

No evento *onClick* do *btnSalvar* inclua a **Listagem 5** a seguir para finalizarmos a codificação das ações dos botões.

Listagem 5. Salvar

```
01 procedure TfrmCadastro.btnSalvarClick(Sender: TObject);
02 begin
03   if Application.MessageBox('Deseja salvar ?', 'Confirmação !',
04     MB_ICONQUESTION + MB_YESNO) = ID_YES then
05     begin
06       dsrCadastro.DataSet.Post;
07       if TClientDataSet(dsrCadastro.DataSet).ApplyUpdates(0)
          <> 0 then
08       begin
09         Application.MessageBox('Problemas ao salvar !',
10             'Confirmação !', MB_ICONERROR + MB_OK);
11         dsrCadastro.DataSet.Refresh;
12       end;
13     end;
14 end;
```

E por último, no *OnClick* do *btnSair* adicione o código a seguir:

```
procedure TfrmCadastro.btnSairClick(Sender: TObject);
begin
  dsrCadastro.DataSet.Close;
end;
```

Selecione o componente *dsrCadastro* e codifique o evento *OnStateChange* para que possamos controlar o estado de cada botão baseado no estado em que se encontra o *DataSet*, como mostra o seguinte código:

```
procedure TfrmCadastro.dsrCadastroStateChange(Sender: TObject);
begin
  btnIncluir.Enabled   := not (dsrCadastro.DataSet.State in
[dsInsert, dsEdit]);
  btnEditar.Enabled    := (not (dsrCadastro.DataSet.State in
[dsInsert])) and (not dsrCadastro.DataSet.IsEmpty);
  btnExcluir.Enabled   := (not dsrCadastro.DataSet.IsEmpty);
  btnCancelar.Enabled  := dsrCadastro.DataSet.State in [dsEdit,
dsInsert];
  btnSalvar.Enabled    := dsrCadastro.DataSet.State in [dsEdit,
dsInsert];
  btnPesquisar.Enabled := (not (dsrCadastro.DataSet.State in
[dsEdit, dsInsert]));
end;
```

Para finalizar o modelo de herança selecione o *frmCadastro* e codifique o evento *OnShow* como mostrado a seguir, deste modo o evento de controle dos botões será automaticamente chamado.

```
procedure TfrmCadastro.FormShow(Sender: TObject);
begin
  dsrCadastroStateChange(Sender);
end;
```

Finalizando, acesse o menu *File>Save all* para salvar o formulário atual e coloque o nome de *untCadastro.pas*, para assim passarmos para a criação da herança de pesquisa.

Vá em *File>New>Form – Delphi for Win32* para que o novo formulário seja criado. Logo, altere suas propriedades *Name* para *frmConsulta*, *Position* para *poMainFormCenter* e também suas medidas de altura e largura setando *Heigth* para *304* e *Width* para *476*. Com isso vamos para os componentes que serão utilizados. Adicione um *ComboBox* e altere suas propriedades *Name* para *cmbPesquisa* e *Style* para *csOwnerDrawFixed*, um *Edit* alterando seu *Name* para *edtPesquisa*, um *DataSource* nomeando-o para *dsrPesquisa*, um *BitBtn* com *Name* de *btnPesquisa* e *Caption* de *Pesquisar*, um *DBGrid* modificando suas propriedades *Align* para *alBottom*, *DataSource* apontando para o *dsrPesquisa* e em *Options* marque para *True* a opção *dgRowSelect*, por último adicione dois objetos *Label* alterando o *Caption* para *Pesquisar por :* e *Dê um duplo clique na Grid para retornar os dados*. Com isso podemos observar o resultado final do *layout* na **Figura 6**.

Adicione à cláusula *Uses* do seu formulário a *unit untDMPrincipal*, para isso utilize o menu *File>Use Unit* ou através do atalho *ALT + F11* e associe à propriedade *DataSet* do *dsrPesquisa*

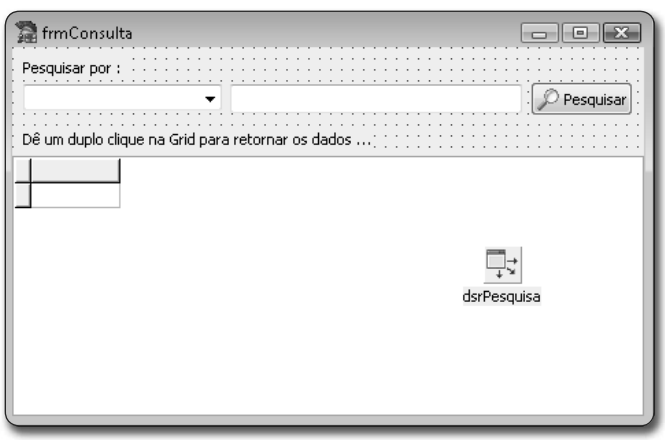


Figura 6. Modelo de formulário de herança para consultas

do *cdsConsulta* localizado no *DMPrincipal*. Partiremos então para a codificação necessária.

Primeiramente vamos fazer com que no ato da destruição do formulário a consulta seja fechada, como mostra o código no evento *OnDestroy* do *frmConsulta:*

```
procedure TfrmConsulta.FormDestroy(Sender: TObject);
begin
  dsrPesquisa.DataSet.Close;
end;
```

Em seguida selecione o componente *cmbPesquisa* e digite a seguinte codificação no evento *OnChange* :

```
procedure TfrmConsulta.cmbPesquisaChange(Sender: TObject);
begin
  edtPesquisa.Clear;
end;
```

Por último, selecione a *DBGrid* e no evento *OnDblClick* faça o tratamento o modo de fechamento do formulário conforme mostrado a seguir:

```
procedure TfrmConsulta.DBGrid1DblClick(Sender: TObject);
begin
  if (dsrPesquisa.DataSet.Active) and (not dsrPesquisa.DataSet.
IsEmpty) then
    Self.ModalResult := mrOk
  else
    Self.ModalResult := mrCancel;
end;
```

Com isso finalizamos toda a parte de herança e podemos finalmente passar para a criação dos formulários restantes da aplicação de exemplo, salve o seu projeto e dê à *unit* o nome de *untConsulta.pas.*

Em nosso exemplo sempre vamos começar criando a consulta e em seguida o cadastro baseado em nossos modelos de herança. Portanto vá em *File>New>Other>Inhretitable Items* e selecione do lado direito o *frmConsulta* para que seja criado o novo formulário. Prossiga alterando seu *Name* para *frmConPessoa* e *Caption* para *Consulta de Pessoas* em seguida selecione o componente *cmbPesquisa* e adicione à sua propriedade *Items Código* e *Nome*. Com isso teremos criado dois parâmetros para consulta, altere seu *ItemIndex* para 0 para que o *Código* seja o primeiro item visível do componente. Agora basta fazer a consulta no evento *OnClick* do *btnPesquisar* conforme mostrado na **Listagem 6.**

Como podemos notar o código da listagem é bem simples, primeiramente é declarada uma constante com a instrução *SQL* previamente definida sem parâmetros, verifica-se então se o conteúdo digitado não está vazio (linha 5), com isso o componente de pesquisa é fechado e os parâmetros são montados e atribuídos dinamicamente conforme o que for escolhido no *ComboBox* (linhas 11 a 23) para que finalmente a pesquisa seja realizada como mostra a linha 24. Salve seu projeto nomeando a *unit* como *untConPessoa.pas.*

Seguindo a seqüência que citei, vamos agora fazer o cadastro. Vá em *File>New>Other>Inheritable Items* e do lado direito selecione o *frmCadastro* para criarmos o Cadastro de Pessoas.

Listagem 6. Evento OnClick do btnPesquisar

```
01  procedure TfrmConPessoa.btnPesquisarClick(Sender: TObject);
02  const
03    SQL = ' SELECT PESSOA.IDPESSOA, PESSOA.NOME FROM PESSOA ';
04  begin
05    if Trim(edtPesquisa.Text) = '' then
06      Application.MessageBox('Nada a pesquisar !', 'Erro !',
         MB_ICONERROR 07  + MB_OK)
08    else
09      begin
10        TClientDataSet(dsrPesquisa.DataSet).Close;
11        case cmbPesquisa.ItemIndex of
12          0:
13            begin
14              TClientDataSet(dsrPesquisa.DataSet).CommandText :=
                 SQL +
15                ' WHERE PESSOA.IDPESSOA = ' + Trim(edtPesquisa.Text);
16            end;
17          1:
18            begin
19              TClientDataSet(dsrPesquisa.DataSet).CommandText :=
                 SQL +
20                ' WHERE PESSOA.NOME LIKE ' +
21              QuotedStr(Trim(edtPesquisa.Text)) +  ' || ' +
                 QuotedStr('%');
22            end;
23        end;
24        TClientDataSet(dsrPesquisa.DataSet).Open;
25      end;
26  end;
```

Inicie alterando sua propriedade *Name* para *frmCadPessoa* e seu *Caption* para *Cadastro de Pessoas*, agora faremos as adaptações necessárias, adicionando ao *Uses* do novo *Form* as Units *DBClient*, *untDMPrincipal* e *untConPessoa*.

Faça a referência do *cdsPessoa* localizado no *DMPrincipal* a propriedade *DataSource* do componente *dsrCadastro* e adicione sete componentes *Label*, seis *DBEdit* e um *DBText* e os ligue ao *dsrCadastro* por meio da propriedade *DataSource*. Por último relacione-os a seus respectivos campos através da propriedade *DataField*.

A única implementação necessária é a chamada da consulta que será comum a todos os outros formulários que iremos criar, como mostra **Listagem 7**.

Listagem 7. Evento OnClick do btnPesquisar

```
01  procedure TfrmCadPessoa.btnPesquisarClick(Sender: TObject);
02  begin
03    frmConPessoa := TfrmConPessoa.Create(Self);
04    with frmConPessoa do
05      try
06        if ShowModal = mrOk then
07          begin
08            dsrCadastro.DataSet.Close;
09            TClientDataSet(dsrCadastro.DataSet).FetchParams;
10            TClientDataSet(dsrCadastro.DataSet).Params.
                ParamByName('IDPESSOA').AsInteger :=
11                dsrPesquisa.DataSet.FieldByName('IDPESSOA').
                   AsInteger;
12            dsrCadastro.DataSet.Open;
13          end;
14      finally
15        FreeAndNil(frmConPessoa);
16      end;
17  end;
```

No código da **Listagem 7** realizamos a chamada da consulta de pessoas e de acordo com o retorno, atribuído de acordo com o clique duplo sobre a *DBGrid*, é feito o carregamento dos parâmetros através do método *FetchParams* (linha 09) e passado como parâmetro o valor da chave primária obtida na consulta (linhas 10 e 11) finalmente é aberto e a consulta é liberada da memória. Com isso salve seu projeto nomeando esta *unit* como *untCadPessoa.pas.*

Partiremos agora para a próxima consulta que refere-se ao cadastro de produtos. Como a implementação é semelhante as já comentadas não entrarei em grandes detalhes, então, do mesmo modo com que criamos a consulta anterior, faça essa acessando o menu *File > New > Other > Inheritable Items > frmConsulta*. Altere o *Name* para *frmConProduto* e o *Caption* para *Consulta de Produtos* e adicione a cláusula *Uses* a Unit *DBClient*. Lembrando que novamente a única codificação necessária é relacionada a formulação da *SQL* de pesquisa e será detalhada na **Listagem 8.** Salve seu projeto e nomeie a nova *unit* com *untConProduto.pas*.

Listagem 8. Evento OnClick do btnPesquisar

```
01  procedure TfrmConProduto.btnPesquisarClick(Sender: TObject);
02  const
03    SQL =
04      ' SELECT PRODUTO.IDPRODUTO, PRODUTO.PRODUTO, '   + #13 +
05      ' PRODUTO.QTD_MIN, PRODUTO.QTD, PRODUTO.VALOR_UNITARIO ' 
          + #13 +
06      ' FROM PRODUTO ';
07  begin
08    if Trim(edtPesquisa.Text) = '' then
09      Application.MessageBox('Nada a pesquisar !', 'Erro !',
         MB_ICONERROR 10  + MB_OK)
11    else
12      begin
13        TClientDataSet(dsrPesquisa.DataSet).Close;
14        case cmbPesquisa.ItemIndex of
15          0: begin
16               TClientDataSet(dsrPesquisa.DataSet).CommandText :=
                  SQL +
17                ' WHERE PRODUTO.IDPRODUTO = ' +
                     Trim(edtPesquisa.Text);
18             end;
19          1: begin
20               TClientDataSet(dsrPesquisa.DataSet).CommandText :=
                  SQL +
21                ' WHERE PRODUTO.PRODUTO LIKE ' +
                  QuotedStr(Trim(edtPesquisa.Text)) + ' || ' +
                  QuotedStr('%');
22             end;
23      end;
24      TClientDataSet(dsrPesquisa.DataSet).Open;
25      end;
26  end;
```

Listagem 9. Evento OnClick do btnPesquisar (Cadastro de Produtos)

```
01  procedure TfrmCadProduto.btnPesquisarClick(Sender: TObject);
02  begin
03    inherited;
04    frmConProduto := TfrmConProduto.Create(Self);
05    with frmConProduto do
06      try
07        if ShowModal = mrOk then
08          begin
09            dsrCadastro.DataSet.Close;
10            TClientDataSet(dsrCadastro.DataSet).FetchParams;
11            TClientDataSet(dsrCadastro.DataSet).Params.
                ParamByName('IDPRODUTO').AsInteger :=
                dsrPesquisa.DataSet.FieldByName('IDPRODUTO').
                 AsInteger;
12            dsrCadastro.DataSet.Open;
13          end;
14      finally
15        FreeAndNil(frmConProduto);
16      end;
17  end;
```

Listagem 10. Evento OnClick do btnPesquisar (Consulta de Vendas)

```
01  procedure TfrmConVenda.btnPesquisarClick(Sender: TObject);
02  const
03    SQL = ' SELECT VENDA.IDVENDA,VENDA.DATA_VENDA FROM VENDA ';
04  begin
05    if Trim(edtPesquisa.Text) = '' then
06      Application.MessageBox('Nada a pesquisar !', 'Erro !',
         MB_ICONERROR+ MB_OK)
07    else
08      begin
09        TClientDataSet(dsrPesquisa.DataSet).Close;
10        case cmbPesquisa.ItemIndex of
11          0:
12            begin
13              TClientDataSet(dsrPesquisa.DataSet).CommandText :=
                  SQL +
14              ' WHERE VENDA.IDVENDA = ' + Trim(edtPesquisa.Text);
15            end;
16          1:
17            begin
18              TClientDataSet(dsrPesquisa.DataSet).CommandText :=
                  SQL +
19                ' WHERE CAST(VENDA.DATA_VENDA AS DATE) = ' +
20                QuotedStr(FormatDateTime('mm/dd/yyyy',
                  StrToDate(edtPesquisa.Text)));
21            end;
21      end;
23      TClientDataSet(dsrPesquisa.DataSet).Open;
24    end;
25  end;
```

Crie o novo formulário de cadastro de produtos em *File >New>Other>Inheritable Items>frmCadastro* e logo altere *Name* para *frmCadProduto* e *Caption* para *Cadastro de Produtos*, adicione à cláusula *Uses* as *units DBClient*, *untDMPrincipal* e *untConProduto*, faça a ligação do componente *cdsProduto*, localizado no *DMPrincipal*, ao *dsrCadastro* na propriedade *DataSource* finalizando assim as configurações. Adicione cinco *Label*, quatro *DBEdit* e um *DBText* novamente configure suas propriedades *DataSource* apontando para o *dsrCadastro* e atribuindo ao *DataField* seus campos correspondentes. Finalize o cadastro implementando a chamada ao formulário de consultas no evento *OnClick* do *btnPesquisar* como mostra a **Listagem 9.** Novamente salve o projeto atribuindo o nome de *untCadProduto.pas*.

Estamos chegando à etapa final do nosso artigo e implementaremos agora a última consulta do projeto. Novamente crie um formulário baseado no *frmConsulta* e altere seu *Name* para *frmConVenda* e *Caption* para *Consulta de Vendas*, adicione à *Uses* a *unit DBClient* e codifique a **Listagem 10** no evento *OnClick* do *btnPesquisar*. Salve esta *unit* com o nome de *untConVenda.pas*.

Antes de prosseguirmos criando o formulário de vendas, vamos voltar nossa atenção para as implementações relacionadas cabíveis a esta etapa, abra o seu *DMPrincipal* e selecione o seu *cdsVenda_Item*, modifique sua propriedade *AggregatesActive* para *True* e dê um duplo clique sobre o mesmo para abrir a janela do *Fields Editor*. Nela crie um novo campo clicando com o botão direito do *mouse* e *New Field*, setando seu *Name* para *TotalItem* e em *Field type* selecione a opção *InternalCalc*, e em seguida repita o processo, porém criando três *Fields* com nomes de *GeraItem*, *TotalVenda* e *QtdTotal* todos de *Field type Aggregate*. Selecione primeiramente os campos do tipo *Aggregate* e altere suas propriedades *Active* e *Visible* para *True* fazendo com que as expressões de agregação sejam calculadas automaticamente pelo *ClientDataSet* e defina a propriedade *Expression* do campo *GeraItem* como *MAX(ITEM)*, para o campo *TotalVenda* defina como *SUM(TotalItem)* e por último *SUM(QUANTIDADE)* para o campo *QtdTotal*.

Para finalizarmos as alterações do *cdsVenda_Item* vamos codificar as últimas ações deste componente, primeiramente vá no evento *OnCalcFields* para realizar o calculo do nosso campo *InternalCalc* como mostrado a seguir:

```
procedure TDMPrincipal.cdsVenda_ItemCalcFields(DataSet: TDataSet);
begin
  if cdsVenda_Item.State in [dsInternalCalc] then
    begin
      cdsVenda_ItemTotalItem.Value :=
        cdsVenda_ItemVALOR_UNITARIO.Value * cdsVenda_ItemQUANTIDADE.
Value;
    end;
end;
```

Em seguida adicione o evento *AfterInsert* para realizar o controle do campo *Item* como exibido no código a seguir:

```
procedure TDMPrincipal.cdsVenda_ItemAfterInsert(DataSet: TDataSet);
begin
  cdsVenda_ItemIDVENDA_ITEM.AsInteger := GetGenerator('IDVENDA_
ITEM');
  if cdsVenda_ItemGeraItem.Value = Null then
    cdsVenda_ItemITEM.AsInteger := 1
  else
    cdsVenda_ItemITEM.AsInteger := cdsVenda_ItemGeraItem.Value + 1;
end;
```

Partiremos para o *cdsVenda*, fazendo a implementação dos dois eventos necessários, sendo o *AfterInsert* que insere os valores requeridos no momento da inclusão da venda:

```
procedure TDMPrincipal.cdsVendaAfterInsert(DataSet: TDataSet);
begin
  cdsVendaIDVENDA.AsInteger := GetGenerator('IDVENDA');
  cdsVendaDATA_VENDA.AsDateTime := Now;
  cdsVendaSTATUS.AsString := 'A';
end;
```

A última alteração é no evento *BeforePost* que apenas atribui o valor do total da venda gerado pelo *aggregate* ao campo *TOTAL_VENDA*:

```
procedure TDMPrincipal.cdsVendaBeforePost(DataSet: TDataSet);
begin
  if cdsVenda.State in [dsEdit, dsInsert] then
    begin
      if cdsVenda_ItemTotalVenda.Value <> Null then
        cdsVendaTOTAL_VENDA.Value := cdsVenda_ItemTotalVenda.Value
      else
        cdsVendaTOTAL_VENDA.Value := 0;
    end;
end;
```

Faremos agora a última alteração do *Data Module* que trata especificamente do tema do nosso artigo, todo o controle transacional da atualização do estoque será realizado pelo componente *DataSetProvider*, vamos então codificar o evento *BeforeUpdateRecord* do nosso *dspVenda* conforme mostra a **Listagem 11.**

Embora esta listagem seja grande o código é bem simples, porém muito interessante. Primeiramente devemos atentar para as linhas 05 a 07 onde temos a definição de duas constantes. Perceba que além do nome, elas fazem tarefas diferentes, as duas serão passadas ao componente *sqlAtualizacao* em tempo de execução e contêm dois parâmetros, sendo eles a quantidade e o *ID* do produto que será atualizado. A diferença é que uma realiza a retirada do estoque subtraindo o valor atual da quantidade recebida por parâmetro, enquanto a outra faz a operação inversa somando a quantidade do parâmetro à quantidade disponível. Como a ocorrência deste evento é realizada de acordo com a quantidade de registros a serem inseridos/modificados, podemos fazer o tratamento exato do controle de estoque. Neste caso o primeiro teste refere-se ao tipo de ação (linha 09), caso seja uma inserção é realizado o segundo teste que verifica se o *DataSet* que está passando pelo evento é o *sqlVenda (mestre)* para assim atribuir o valor *F (venda finalizada)* para o campo *STATUS*. Caso contrário, o *sqlAtualizacao* é fechado, a instrução *SQL* é passada ao mesmo e os parâmetros são atribuídos realizando assim a retirada do estoque (linhas 13 a 20). Na linha 22 é feito o segundo teste condicional relativa à ação a ser tomada, sendo uma atualização, a respectiva constante é atribuída e os parâmetros novamente passados, porém, resgatando os valores antigos através do método *OldValue*, para assim atualizarmos os produtos corretamente como podemos observar entre as linhas 25 a 31.

Agora que finalizamos as modificações necessárias no *Data Module* vamos finalizar o artigo criando a tela de venda e fazendo a chamada final dos formulários. Crie um novo formulário baseado no *frmCadastro* e altere suas propriedades *Name* para *frmCadVenda* e *Caption* para *Vendas*, altere a disposição de altura e largura conforme seu gosto, para o exemplo utilizei *Height* com *509* e *Width* de *684*, faça as modificações relativas ao *Uses* incluindo as *units DBClient*, *untConPessoa*, *untConProduto*, *untDMPrincipal* e *untConVenda*.

Listagem 11. Evento BeforeUpdateRecord do componente dspVenda

```
01  procedure TDMPrincipal.dspVendaBeforeUpdateRecord(Sender: TObject;
02   SourceDS: TDataSet; DeltaDS: TCustomClientDataSet; UpdateKind:
03  TUpdateKind;
04    var Applied: Boolean);
05  const
06    SQL_RETIRA_ESTOQUE =' UPDATE PRODUTO SET PRODUTO.
       QTD = PRODUTO.QTD - :QUANTIDADE ' +' WHERE PRODUTO.
        IDPRODUTO =:IDPRODUTO ';
07    SQL_DEVOLVE_ESTOQUE =' UPDATE PRODUTO SET PRODUTO.
       QTD = PRODUTO.QTD + :QUANTIDADE ' +' WHERE PRODUTO.
        IDPRODUTO =:IDPRODUTO ';
08  begin
09    if UpdateKind = ukInsert then
10      begin
11        if SourceDS = sqlVenda then
12          DeltaDS.FieldByName('STATUS').NewValue := 'F'
13        else if SourceDS = sqlVenda_Item then
14          begin
15            sqlAtualizacao.Close;
16            sqlAtualizacao.CommandText := SQL_RETIRA_ESTOQUE;
17            sqlAtualizacao.Params[0].Value :=
                DeltaDS.FieldByName('QUANTIDADE').Value;
18            sqlAtualizacao.Params[1].Value :=
                DeltaDS.FieldByName('IDPRODUTO').Value;
19            sqlAtualizacao.ExecSQL;
20          end;
21      end
22    else if UpdateKind = ukModify then
23      begin
24        if SourceDS = sqlVenda_Item then
25          begin
26            sqlAtualizacao.Close;
27            sqlAtualizacao.CommandText := SQL_DEVOLVE_ESTOQUE;
28            sqlAtualizacao.Params[0].Value :=
                DeltaDS.FieldByName('QUANTIDADE').OldValue;
29            sqlAtualizacao.Params[1].Value :=
                DeltaDS.FieldByName('IDPRODUTO').OldValue;
30            sqlAtualizacao.ExecSQL;
31          end;
32      end;
33  end;
```


ClubeDelphi PLUS www.devmedia.com.br/clubedelphi/portal.asp

Acesse agora mesmo o portal do assinante ClubeDelphi e veja uma vídeo aula de Everson Borges Volaco que mostra como utilizar campos Aggregates no ClientDataSet.

http://www.devmedia.com.br/articles/viewcomp.asp?comp=1376

Nota

Para que a transação seja reaproveitada, é fundamental que o componente que realizará atualização participe da mesma conexão que está sendo utilizada pelo DataSetProvider. Para isso basta certificar com que esteja ligada ao mesmo SQLConnection.

Primeiramente ligue o *dsrCadastro* ao *cdsVenda* localizado *DMPrincipal* através de sua propriedade *DataSource*. Em seguida, adicione ao seu formulário cinco *BitBtn* modificando suas propriedades *Name* respectivamente para *btnAdicionarItem, btnExcluirItem, btnPesquisaPessoa, btnPesquisaProduto* e *btnCancelaVenda* e *Caption* para *Adicionar Item, Excluir Item, Consulta Pessoa, Consulta Produto e Cancelar Venda*, adicione também um *DataSource* alterando o *Name* para *dsrItem* e vincule-o ao *cdsVenda_Item* localizado no *DMPrincipal*, um *Panel* modificando sua propriedade *Align* para *alBottom*, um *DBGrid* modificando seu *Align* para *alBottom, DataSource* para *dsrItem* e em *Options* ative a opção *dgRowSelect*, adicione um *DBEdit* e ligue sua propriedade *DataSource* ao *dsrItem*, atribua ao *DataSetField* o campo *QUANTIDADE*, adicione os componentes restantes de acordo com os campos da venda e do Item da Venda (*DBGrid*). Para o exemplo utilizei apenas componentes *DBText* e *Label*. Veja o layout final de nossa tela na **Figura 7.**

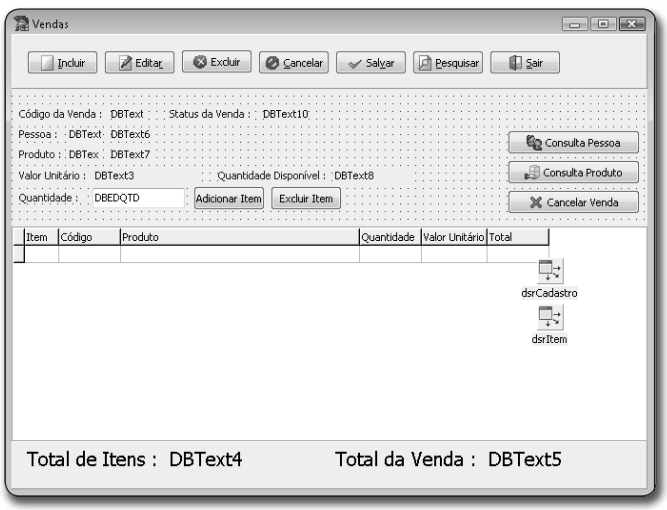


Figura 7. Tela de vendas finalizada

Partiremos para as implementações finais iniciando pelo evento *FormShow* que ocultará os botões de edição e exclusão não permitindo estas ações no nosso formulário :

```
procedure TfrmCadVenda.FormShow(Sender: TObject);
begin
  inherited;
  btnEditar.Visible := False;
  btnExcluir.Visible := False;
end;
```

Em seguida adicione o código a seguir ao evento *OnStateChange* do componente *dsrCadastro*, para controlar o estado do botão de cancelamento da venda:

```
procedure TfrmCadVenda.dsrCadastroStateChange(Sender: TObject);
begin
inherited;
  btnCancelaVenda.Enabled :=
    dsrCadastro.DataSet.FieldByName('STATUS').AsString = 'F';
end;
```

Acrescente o seguinte código ao evento *OnClick* do botão *btnExcluirItem*:

```
procedure TfrmCadVenda.btnExcluirItemClick(Sender: TObject);
begin
  inherited;
  if dsrCadastro.DataSet.FieldByName('STATUS').AsString = 'A' then
    begin
      if Application.MessageBox('Deseja excluir este item ?',
        'Confirmação !', MB_ICONQUESTION + MB_YESNO) = ID_YES then
          dsrItem.DataSet.Delete;
    end;
end;
```

Na **Listagem 12** temos o evento *OnClick* do *btnAdicionarItem*, onde é verificado se o estado do *DataSet* é de inserção ou edição e se o campo *STATUS* está como *A (venda em andamento)*, sendo assim verifica se o *DataSet* relativo aos itens também está nos mesmos estados, se a quantidade digitada for maior que 0 e não nula o item é então gravado, caso contrário a mensagem de erro é exibida.

Listagem 12. Evento OnClick do btnAdicionarItem

```
01  procedure TfrmCadVenda.btnAdicionarItemClick(Sender: TObject);
02  begin
03    inherited;
04    if (dsrCadastro.DataSet.State in [dsInsert, dsEdit]) and
05       (dsrCadastro.DataSet.FieldByName('STATUS').AsString = 'A')
          then
06      begin
07        if dsrItem.DataSet.State in [dsEdit, dsInsert] then
08          begin
09            if (dsrItem.DataSet.FieldByName('QUANTIDADE').
                 Value <> Null)
              and (dsrItem.DataSet.FieldByName('QUANTIDADE').
                 Value > 0) then
10               dsrItem.DataSet.Post
11            else
12              Application.MessageBox('Quantidade inválida !',
                  'Erro !',
              MB_ICONERROR + MB_OK);
13          end;
14      end;
15  end;
```

A outra parte fundamental para o nosso controle transacional é o evento *OnClick* do botão *btnCancelaVenda* que pode ser visualizado na **Listagem 13.**

Como podemos perceber é pedida a confirmação do cancelamento (linha 6), e caso seja confirmada o campo *STATUS* recebe o valor *C* (venda cancelada) para que em seguida sejam percorridos os itens atribuindo o valor *F (falso)* para o campo *ATIVO* localizado no *cdsVenda_Item* (linhas 12 a 18) e por último as aplicações serão feitas (linha 22) exibindo uma mensagem de sucesso em caso positivo ou erro caso contrário.

Por último, exibiremos na **Listagem 14** as referências de chamadas dos formulários que deverão ser adicionados a cláusula *Uses* do *frmMenu* e implementadas em seus itens correspondentes no componente *MainMenu*. Após finalizar acesse o menu *File>Project>Options>Forms* e na opção *Auto-create forms* deixe apenas os formulários *frmMenu* e *DMPrincipal*. Salve seu projeto nomeando a *unit* restante como *untCadVenda* para assim concluir o nosso artigo.

Listagem 13. Evento OnClick do btnCancelaVenda

```
01  procedure TfrmCadVenda.btnCancelaVendaClick(Sender: TObject);
02  begin
03    inherited;
04    if dsrCadastro.DataSet.Active then
05      begin
06        if Application.MessageBox('Deseja cancelar a venda ?',
            'Confirmação !', MB_ICONQUESTION + MB_YESNO) =
              ID_YES then
07          begin
08            dsrCadastro.DataSet.Edit;
09            dsrCadastro.DataSet.FieldByName('STATUS').AsString := 'C';
10            dsrItem.DataSet.DisableControls;
11            dsrItem.DataSet.First;
12            while not dsrItem.DataSet.Eof do
13              begin
14                dsrItem.DataSet.Edit;
15                dsrItem.DataSet.FieldByName('ATIVO').AsString := 'F';
16                dsrItem.DataSet.Post;
17                dsrItem.DataSet.Next;
18              end;
19            dsrCadastro.DataSet.Post;
20            dsrItem.DataSet.First;
21            dsrItem.DataSet.EnableControls;
22            if TClientDataSet(dsrCadastro.DataSet).
                ApplyUpdates(0) = 0 then
23              Application.MessageBox('Venda cancelada com sucesso !',
                  'Confirmação !', MB_ICONINFORMATION + MB_OK)
24            else
25              Application.MessageBox('Problemas ao cancelar a
                  venda !',
                    'Erro !', MB_ICONERROR + MB_OK);
26          end;
27      end;
28  end;
```

Nota

Devido a extensão do artigo, os códigos relativos às chamadas das consultas de Pessoas, Produtos e vendas não serão mostrados aqui, porém, o exemplo estará disponível para download no site da DevMedia.

Listagem 14. Chamadas dos formulários no MenuPrincipal

```
01 [Cadastro de Pessoas]
02 procedure TfrmMenu.Pessoas1Click(Sender: TObject);
03 begin
04   frmCadPessoa := TfrmCadPessoa.Create(Self);
05   with frmCadPessoa do
06     try
07       ShowModal;
08     finally
09       FreeAndNil(frmCadPessoa);
10     end;
11 end;

12 [Cadastro de Produtos]
13 procedure TfrmMenu.Produtos1Click(Sender: TObject);
14 begin
15   frmCadProduto := TfrmCadProduto.Create(Self);
16   with frmCadProduto do
17     try
18       ShowModal;
19     finally
20       FreeAndNil(frmCadProduto);
21     end;
22 end;

23 [Sair]
24 procedure TfrmMenu.Sair2Click(Sender: TObject);
25 begin
26   Self.Close;
27 end;

28 [Vendas]
29 procedure TfrmMenu.Vendas2Click(Sender: TObject);
30 begin
31   frmCadVenda := TfrmCadVenda.Create(Self);
32   with frmCadVenda do
33     try
34       ShowModal;
35     finally
36       FreeAndNil(frmCadVenda);
37     end;
38 end;
```

Conclusão

Como vimos existem várias maneiras de se controlar transações, porém sejam como a prática apresentada neste artigo ou de forma manual, estas devem ser controladas de forma cuidadosa para evitar possíveis problemas no controle de concorrência do seu *software*. Mais uma vez espero que tenham gostado, um forte abraço e nos vemos no próximo artigo. ●

Dê seu feedback sobre esta edição!

A Clubedelphi tem que ser feita ao seu gosto. Para isso, precisamos saber o que você, leitor, acha da revista!

Dê seu voto sobre este artigo, através do link:

www.devmedia.com.br/clubedelphi/feedback

***Sr. Cliente favor ler as informações com atenção, dúvidas entrar em contato com atendimento.**

Relatório de Pré-Flight

OP: Cliente: Neoficio Data: 07/05/09 Operador: JULIO

Informações da OP:
Título:
Quantidade de Páginas:
Formato do Miolo:
Formato da Capa:
Tamanho da Lombada:
Tipo de Acabamento:

Informações do Arquivo:
Título: Clube Delphi
Quantidade de Páginas: 64 + Capa
Formato do Miolo: 205x275
Formato da Capa: 205x275
Tamanho da Lombada:------
Tipo de Acabamento: Canoa

Informações do Arquivo

• Imagens em baixa resolução com menos de 200 DPI:
Páginas: Todo o Miolo
• Texto preto em quatro cores nas páginas:
Páginas: 02
• Imagens em RGB nas páginas:
Páginas: Capa-02

• Colofon / Tiragem: ( ) OK ( ) Não
• Código de Barras: (x) OK ( ) Não
• Páginas sem sangria e/ou inferior a 3 mm (Filete branco no acabamento):
Páginas:
• Texto rente ao corte nas páginas:
Páginas:

OBS: As informações em **VERMELHO** necessitam de atenção, se de acordo favor assinar.

Data ____/____/____ Cliente: _____

Nome Legível

Todas as Provas de Cor devem ser assinadas de acordo com a aprovação. Havendo alteração na cor favor anotar, e será enviado nova Prova de Cor.

Observações: